



Linnæus University

School of Computer Science, Physics and Mathematics

Degree Project

Data Encryption on a Network

Ignacio Arechaga Fernandez
Jorge Luque González
2010-11-14
Subject: Computer Science
Level: Bachelor
Course code: 2DV00E

Data Encryption on a Network

Ignacio Arechaga Fernandez
Jorge Luque Gonzalez

September 2010

Abstract

In this project you can find a study about different encryption algorithms, which are use to safeguard the information on messages over the network. We have developed a client-server application which will send information throught the network which has to be secured. There are two kinds of encryption algorithms, the symmetric and the asymmetric key algorithms. Both were used to establish the communication, the asymmetric algorithm (*RSA*) isi used to set up a symmetric key and then, all the communication process is done only with the symmetric algorithm (*Blowfish*).

Keywords : encryption, decryption, DES, Triple-DES, security, serpent, blow-fish, Diffie-Hellman, RSA, AES,private key, public key, security, attack, brute force, man-in-the-middle.

Contents

1	Introduction	1
1.1	Problem definition	1
1.2	Report structure	2
2	Implementation	3
2.1	Functional requirements	3
2.2	Non functional requirements	3
2.2.1	Usability	3
2.2.2	Reliabilitly	4
2.2.3	Efficiency	4
3	Algorithms	5
3.1	Data Encryption Standard (DES)	5
3.1.1	Theoretical description	5
3.1.2	Algorithm's security	7
3.1.3	Practice analysis	9
3.2	Triple Data Encryption Standard (Triple-DES)	11
3.2.1	Theoretical description	11
3.2.2	Algorithm's security	12
3.2.3	Practice analysis	13
3.3	Advanced Encryption Standard (AES)	15
3.3.1	Theoretical description	15
3.3.2	Algorithm's security	16
3.3.3	Practice analysis	17
3.4	Blowfish	22
3.4.1	Theoretical description	22
3.4.2	Algorithm's security	23
3.4.3	Practice analysis	24
3.5	Serpent	27
3.5.1	Theoretical description	27
3.5.2	Algorithm's security	29
3.5.3	Practice analysis	29
3.6	Rivest Shamir Adleman (RSA)	31
3.6.1	Theoretical description	31
3.6.2	Algorithm's security	31
3.6.3	Practice analysis	32
3.7	Diffie - Hellman key exchange	34
3.7.1	Theoretical description	34
3.7.2	Algorithm's security	35
3.7.3	Practice analysis	36
4	Discussion	37
5	Java study application	42
5.1	Class Diagram	42
5.2	Package Hierachy	44
5.2.1	Default Package	44
5.2.2	Model Package	44

5.2.3	Controlador Package	44
5.2.4	Vista Package	44
5.2.5	Observador Package	44
5.3	User Manual	45
5.3.1	Main Window	45
5.3.2	Result Window	45
6	Client-Server application	47
6.1	Class diagrams	47
6.1.1	Client Diagrams	47
6.1.2	Server Diagrams	52
6.2	Server	58
6.2.1	Aplicacion Package	58
6.2.2	Modelo Package	58
6.2.3	Controlador Package	59
6.2.4	Observador Package	59
6.2.5	Vista Package	59
6.2.6	Hilo Package	60
6.3	Client	61
6.3.1	Modelo Package	61
6.3.2	Controlador Package	62
6.3.3	Observador Package	62
6.3.4	Vista Package	62
6.3.5	Hilo Package	63
6.4	Resources used	64
6.5	Encryption implementation on the applications	64
6.5.1	Client application :	64
6.5.2	Server application :	67
6.5.3	Common code between both applications :	69
6.5.4	Security problems found during the development	71
7	Conclusion	72
8	Future Work	73
	References	74
A	User Manual	77
A.1	Login Windows	77
A.2	Main window : Restaurants	78
A.3	Main window : Clients	84
A.4	Main window : Restaurants and Clients	86

1 Introduction

This thesis has been written by Jorge Luque Gonzalez and Ignacio Arechaga Fernandez. It's supervised by Tobias Andersson-Gidlund from Linnæus University. This is our Bachelor Degree Project in Computer Science for Linnæus University.

The encryption is the process of transforming information using an algorithm to make it unreadable to anyone except those possessing special knowledge. To do it we need a key (a piece of information that determines the functional output of a cryptographic algorithm or cipher).

The encryption is used by armies, governments, banks, companies, etc. Encryption can be used to protect data "at rest", such as files on computers and storage devices or to protect data in transit, for example data being transferred via networks.

Since there are many different encryption algorithms, we want study them and choose one which satisfies the desired requirements for our client-server application. There are 4 points in common between all the different encryption algorithms :

- Plain text : Which is the message we want to send to the destination. It is human readable or formatted in a way that a specific application is able to read it.
- Ciphered text : The modified plain text and it cannot be read until is deciphered.
- Encryption algorithm : The way to process the data to make it unreadable.
- Key : A specific key to cipher and/or decipher the plain text or the ciphered text.

1.1 Problem definition

In this report we will do a study of some of the most known encryption algorithms and we will compare them. We are going to build an application which studies the time consumption of the different algorithms on encryption and decryption. We are also going to develop a Java application which needs access to a database. This connection is done over the network, therefore we will need to protect the passwords and the information transferred to increase the security and the user privacy.

Encryption on computers have been used since a long time ago. There are many different types of algorithms for encrypt a message. This algorithms would be changed for others because they become obsolete over the time. Computers get faster and powerful and as a result we need stronger algorithms.

Encryption takes a big relevance with the Internet. You may have to send some information over the network on a non secure environment. This message can be intercepted by others who are not supposed to receive it, therefore you can have your information under risk.

To encrypt a message we have two main algorithm categories : asymmetric and symmetric encryption algorithms.

Symmetric algorithms : The sender encrypts the message with a key and the same key is used to decipher it. These algorithms are fast and they can encrypt and decrypt efficiently with relatively large keys. The problem here is the security of the key. We need to send the encrypted message so that

nobody can decrypt it without the key. However, the receiver must know the key, how do you get it without compromising their security?

Asymmetric algorithms : These algorithms have two different keys. One is used to encrypt the messages and the other one to decrypt them. Therefore we can talk of both keys as a *public key* (to encrypt) and a *private key* (to decrypt). This ensures confidentiality, anyone can encrypt, but only who has the private key can decrypt the original message.

1.2 Report structure

This report has been structured by chapters. We have divided the report in 7 chapters and one appendix. Before each chapter there is a short introduction explaining how each one is structured.

Chapter 1 : Introduction, where we explain the problems approached and how we will resolve them.

Chapter 2 : Implementation, where we discuss our programming language choice for the cypher algorithms.

Chapter 3 : Algorithms, where we are going to present each algorithm, explain the theory of the algorithm, the security capabilities and the algorithm encryption and decryption performance.

Chapter 4 : Discussion, where we will analyze all the algorithms together and we will choose the one which fits our requirements.

Chapter 5 : Java Study Application, where we present the application built to test and study the algorithms.

Chapter 6 : Client-Server Application, where we introduce the client server application design and the application manual.

Chapter 7 : Conclusion, where we summarize all our conclusions of this project.

Chapter 8 : Future Work, where we suggest some further work from this project.

2 Implementation

Our client-server application is written in Java. The reason of this choice is because we are supplying an application which needs low resources, therefore it can be run in any computer. With this hardware requirements, the clients and servers will not need to change their computer system to allocate the new application.

Another advantage is the OS (*Operative System*) independency of Java.

The implementation language of the algorithms should not matter in theory. In practice, it does matter. That is because some algorithms are more *compiler friendly*. This means that some algorithms can be optimized during compilation time, others do not. Therefore those with that optimization will be faster and we should consider this during our algorithm study.

2.1 Functional requirements

Application to compare algorithms : We want to develop an application which will measure the times spend to encrypt, and decrypt, will compare the size of the key and the size of the original and the encrypted message. The user will write the message to encrypt and choose the type of encryption.

In the restaurant application, we have to separate the client (user) and the server (restaurant). The communication between the server and the client must be encrypted.

The client is the user that wants buy food on someone of our restaurants. He can choose the food he wants on our restaurants and after this he must be able to order it. The clients will be able to look for food by restaurants or by type of food. They will also be able to see their orders. The application will provide a chat where the clients and the restaurants could talk; the clients only can talk on the general room or on the rooms which the administrators (restaurants) have created.

The restaurants are the servers. They can see and modify all the information on the database. They can talk with the clients on the chat and create/delete chat rooms. They can see some statistics like their biggest orders, by customer orders, average order per customer, sell units and their price, and clients with orders up than 150 €. They can also see a client list, products and orders per month.

2.2 Non functional requirements

The no functional requirements are the requirements which do not have a directly relation with the service that the application must provide. These requirements define the system behaviour such as timing, or aspects as system speed, performance, etc.

2.2.1 Usability

According to [1] a high *usability* will allow the new clients and servers to use our software system without expending much resources (time and money) to train their employees. Therefore we must design and build an user-friendly application, easy to navigate through all the available options on the system.

2.2.2 Reliability

According to [1] *reliability* determines the maximum allowed software system failure rate. Since it is an important quality factor and here we are dealing with sensitive information, this quality factor must reach a high level.

2.2.3 Efficiency

A high *efficiency* means that the hardware requirements to run our application aren't so high. That means that the computer's processing capabilities, memory and disk capacity and the data communication capability of the network is fairly enough for our running application [1]. Since it is Java based, the operative system should not be fixed, allowing the clients and servers to use the one they already have.

Since we are going to test different encryption algorithms, we should be careful on this quality factor. The different algorithms have different resource consumptions as they can be more complex than others.

3 Algorithms

In this chapter we will be presenting each encryption algorithm. First we will do a theoretical study of each one, followed by a security and analysis. After that, we will show the results of our practical analysis.

3.1 Data Encryption Standard (DES)

Data Encryption Standard is an encryption algorithm designed by IBM. It was chosen as a standard on 1976 by the *National Bureau of Standards* as an official *Federal Information Processin Standard (FIPS)* from United States.

The algorithm uses a key size of 64 bits and encrypts blocks of 64 bits. The key size is 64 bits but only 56 bits are used by the algorithm (to encrypt data). The other 8 bits are used for parity checking and they are discarded after this. This algorithm is too old and with the time the size of the key fell short.

3.1.1 Theoretical description

There are 16 identical rounds (stages of processing). There is an initial (IP) and a final (FP) permutation, that are inverses (IP "undoes" the action of FP, and vice versa).

The block is divided into two 32-bit halves, before the main rounds, and processed alternately (this criss-crossing is known as the *Feistel scheme*) as is shown in Figure 3.1.1.

The Feistel structure ensures that decryption and encryption are very similar processes (the difference is that the sub-keys are applied in the reverse order when decrypting). The red symbol (on the figure) denotes the XOR operation. The Feistel function scrambles half a block together with one sub-key. The output from the Feistel function is then combined with the other half of the block, using a XOR operation (The red symbol on the figure denotes the XOR operation), and the halves are swapped before the next round. On the final round, the halves are not swapped.

The Feistel function (F): operates on half a block (32 bits) at a time and consists of four stages:[4, 5]

1. Expansion : the 32-bits are expanded to 48 bits using the *expansion permutation* (E), duplicating half of the bits. The output consists of eight 6-bits pieces, each containing a copy of 4 corresponding input bits, plus a copy of the immediately adjacent bit from each of the input pieces to either side.
2. Key mixing : the result of the last stage is combined with a *sub-key* using an XOR operation.
3. Substitution : The block is divided into eight 6-bits pieces before processing by the substitution boxes (*S-boxes*). S-box stands for *Substitution-box* and is a basic component of symmetric key algorithms which performs substitution. They're used to make a blur relation between the key and the ciphertext. One example can be found in Figure 3.1.4.

The eight substitution boxes replace its six input bits with four output bits according to a non-linear transformation, provided in the form of a lookup table.

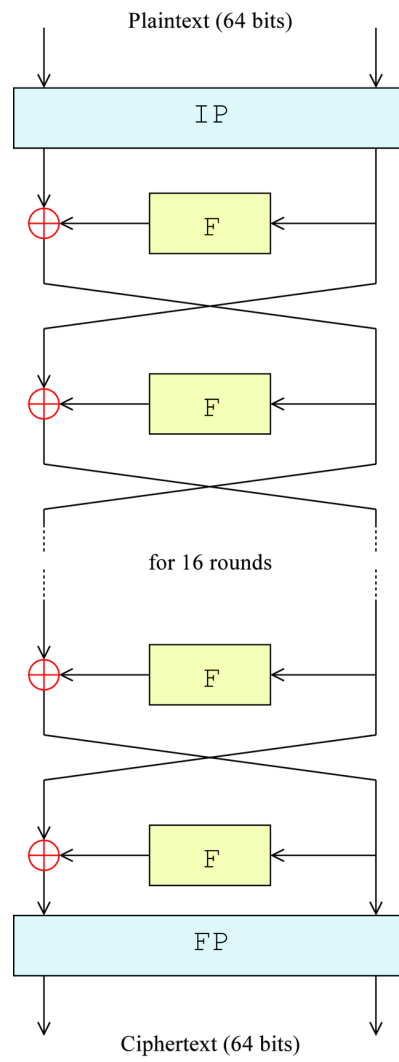


Figure 3.1.1: Data Encryption Standard — Feistel structure[36]

4. Permutation : At the end, the 32 outputs from the substitution boxes are rearranged according to a fixed permutation.

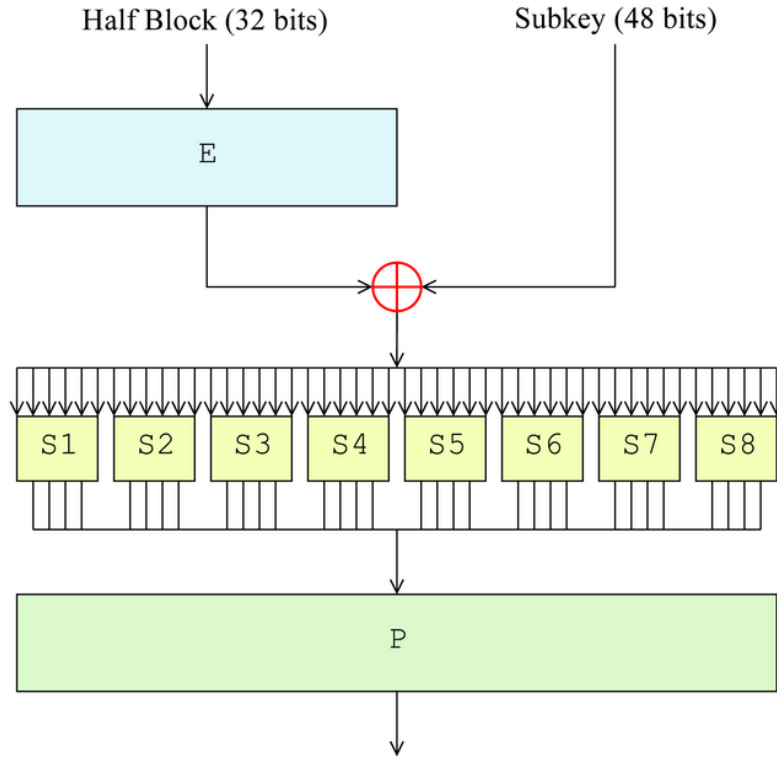


Figure 3.1.2: Data Encryption Standard — key schedule[37]

The key schedule is shown in Figure 3.1.2. The 56 bits of the key are selected from the initial 64 bits by Permuted Choice1 (PC-1) (Figure 3.1.3) the remaining bits are discarded as parity check bits. The 56 bits are divided into two halves; each half is thereafter treated separately. In successive rounds, both halves are rotated left one or two bits (specified for each round), and then 48 sub-key bits are selected by Permuted Choice 2 (PC-2), 24 from the left, and 24 from the right.

3.1.2 Algorithm's security

This algorithm have a key of 64 bits but only 56 are effective, therefore 2^{56} different keys are possible.

Today the best way to attack the DES algorithm is by using *brute force* (trying every possible key), although there are some other theoretical attacks. Their theoretical complexity is less than the brute force attack, therefore they require much less time to break it. *Differential cryptanalysis*[17] requires 2^{47} chosen plaintexts to break the full 16 rounds. *Linear cryptanalysis*[18] needs 2^{43} knowns plaintexts. *Improved Davies' attack*[19] presented another attack which requires 2^{52} known plaintexts.

Therefore, this algorithm is no longer secure enough.

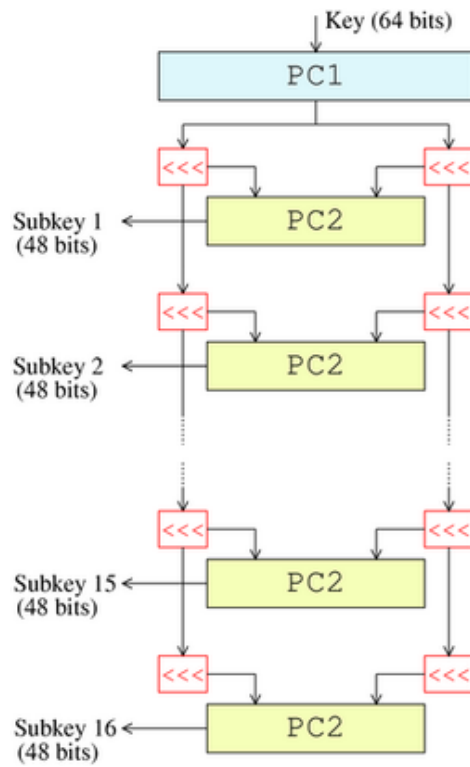


Figure 3.1.3: Data Encryption Standard — Permuted Choice[38]

S_5		Middle 4 bits of input															
		0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Outer bits	00	0010	1100	0100	0001	0111	1010	1011	0110	1000	0101	0011	1111	1101	0000	1110	1001
	01	1110	1011	0010	1100	0100	0111	1101	0001	0101	0000	1111	1010	0011	1001	1000	0110
	10	0100	0010	0001	1011	1010	1101	0111	1000	1111	1001	1100	0101	0110	0011	0000	1110
	11	1011	1000	1100	0111	0001	1110	0010	1101	0110	1111	0000	1001	1010	0100	0101	0011

Figure 3.1.4: Example of S-Box[35]

3.1.3 Practice analysis

In our practical work, we have measured the time consumption during the encryption and decryption. Since the OS eventually expropriates the CPU, we have done rounds of 100.000 and 1.000.000 repetitions and then, we show the average of them.

The measured times are the following :

Encryption :			
Message Size	Encrypted Message Size	100.000 repetitions	1.000.000 repetitions
56 bits	64 bits	1167.71 ns	1167.25 ns
64 bits	128 bits	1692.19 ns	1686.31 ns
128 bits	192 bits	2080.54 ns	2068.97 ns
256 bits	320 bits	2970.91 ns	2967.44 ns
512 bits	576 bits	4878.58 ns	4860.44 ns
1024 bits	1088 bits	8373.25 ns	8331.23 ns

Decryption :		
Message Size	100.000 repetitions	1.000.000 repetitions
56 bits	1200.42 ns	1196.53 ns
64 bits	1678.74 ns	1681.93 ns
128 bits	2173.51 ns	2159.45 ns
256 bits	3127.23 ns	3108.69 ns
512 bits	4959.05 ns	4948.22 ns
1024 bits	8678.04 ns	8646.15 ns

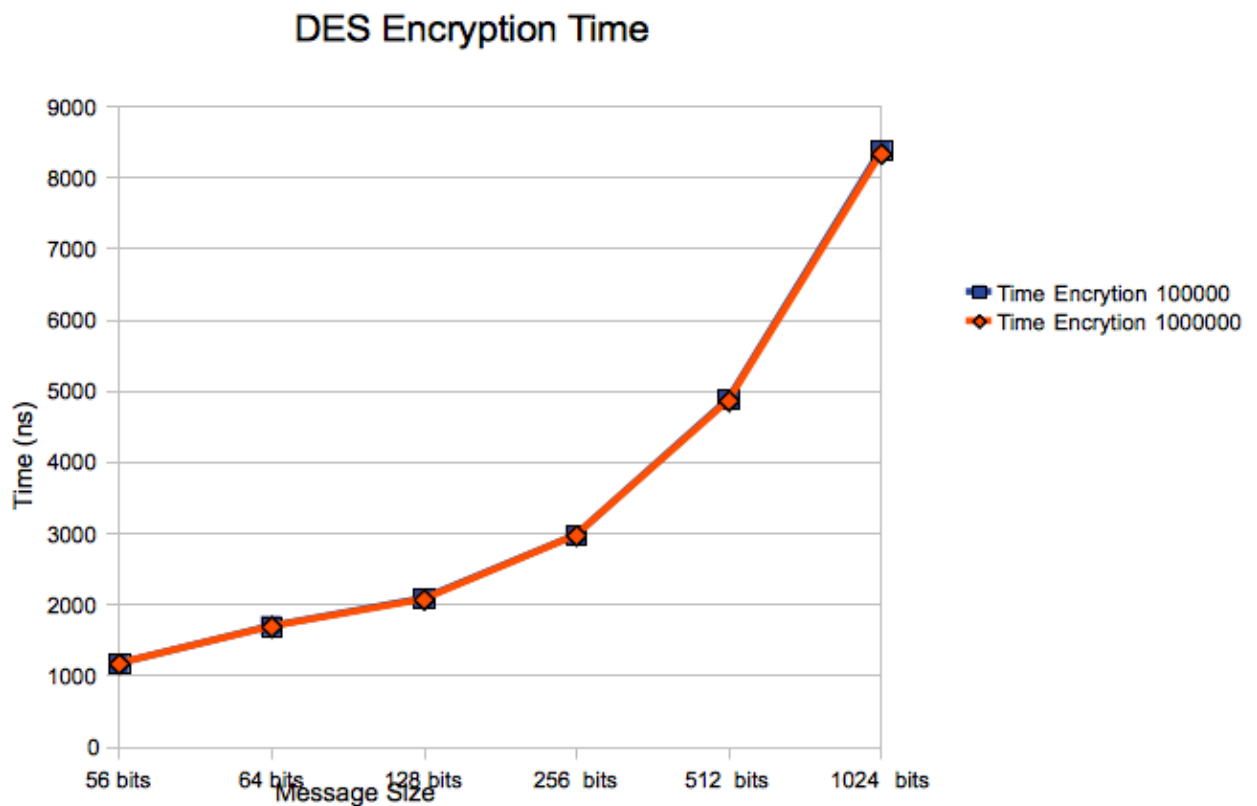


Figure 3.1.5: DES Encryption

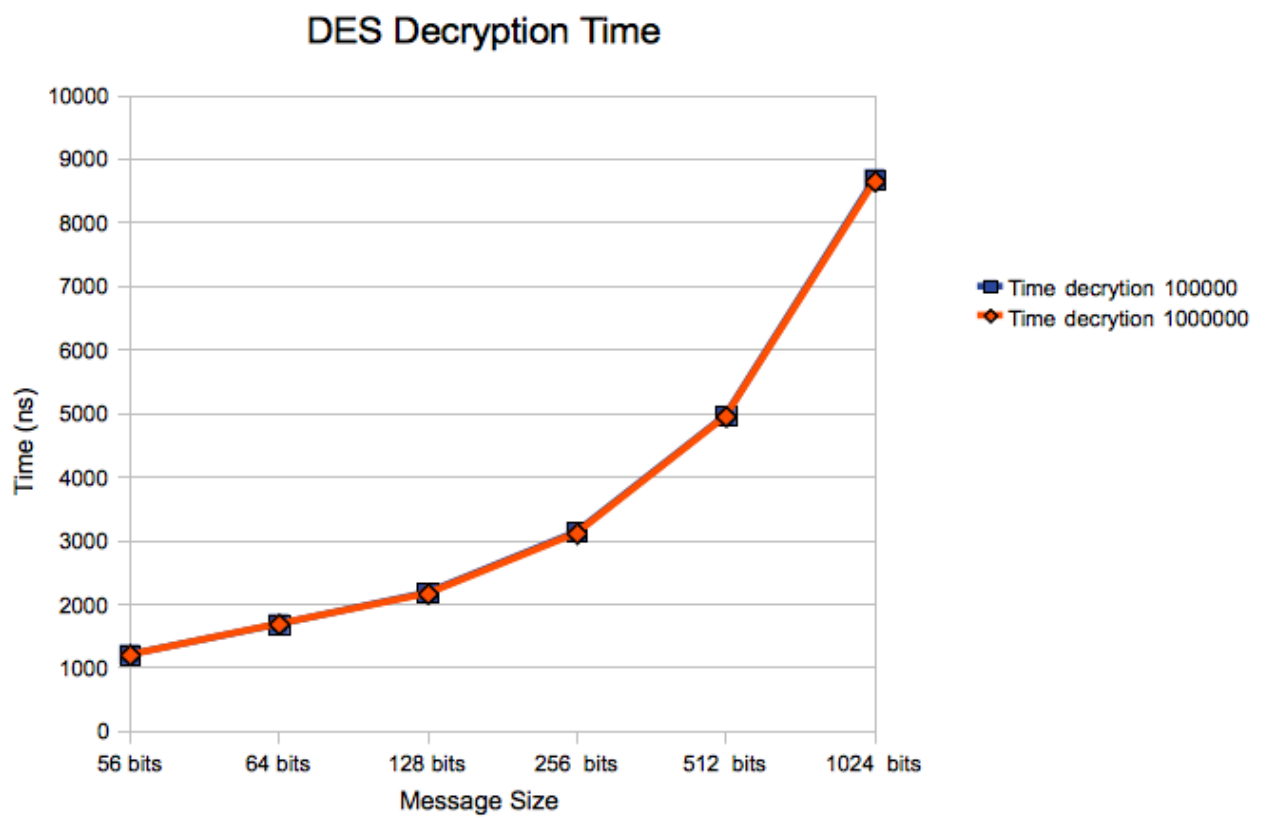


Figure 3.1.6: DES Decryption

3.2 Triple Data Encryption Standard (Triple-DES)

As we can see, *DES* uses relatively short key length. *Triple-DES*[6] uses the *DES* encryption but instead of encrypting one time with one key, it uses 3 different encryptions with 3 different keys. It was developed by *Walter Tuchman* in 1998. The main purpose is to increase the key length so that brute force attacks become impossible[7].

3.2.1 Theoretical description

A key in *Triple-DES* is a triple $K = (K_1, K_2, K_3)$ where each K_i is a DES key. *DES* uses a 56 bit key, therefore *Triple-DES*'s key has $3 * 56 = 168$ bits.[6]

Since there is no much difference between encryption and decryption in *DES*, it is usual to replace the second encryption with a decryption (Figure 3.2.1) in *Triple-DES*. This decryption does not affect the security but it allows retrocompatibility with *DES*.

If we set K_1, K_2 and K_3 as keys, and $K_1 = K_2 = K_3$, we have the same effect as the original *DES* (the decryption is neutralized with the second encryption). The encryption is denoted by $DES_{K_i}(m)$ and the decryption is denoted by $DES_{K_i}^{-1}(m)$.

$$C = DES_{K_3}(DES_{K_2}^{-1}(DES_{K_1}(M)))$$

The message is encrypted with DES and the key K_1 , then 'decrypted' with DES and the key K_2 and then encrypted again using DES and the key K_3

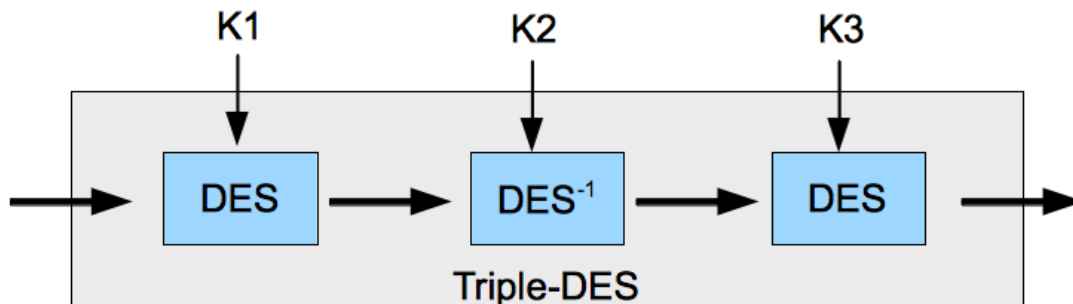


Figure 3.2.1: Triple-Data Encryption Standard

For this algorithm is not absolutely necessary to have three different keys (Figure 3.2.2). We can have $K_1 = K_3$ and then the key length would become 112 bits. Shorter but long enough. In practice, Triple-DES is usually used with only two keys[6].

An important property of DES exploited by Triple-DES is that it does not form a group with the keys (thanks to the 'decryption' with K_2). If we only had encryptions, it would be equivalent as only one DES encryption with a different key, therefore Triple-DES wouldn't be more secure than DES.

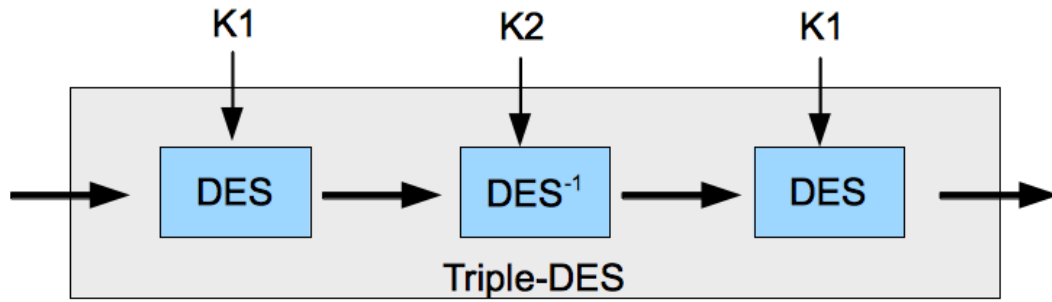


Figure 3.2.2: Triple-Data Encryption Standard with $K_1 = K_3$

3.2.2 Algorithm's security

3DES is a algorithm which is based on DES, this algorithm uses three keys of the DES algorithm and each one of this keys are used with the DES algorithm independently, so 3DES is the result of apply the DES algorithm three times and like DES encrypt 8 bytes of data[28].

DES has a key size of 64 bits, but 8 of these bits are of parity checking so the effective size of the key is 56; assuming that 3DES has three DES keys we can see that the maximum size of the effective key will be 168 bits (56 bits x 3), however it has 192 bits of long (64 bits x 3).

1. 3 different keys : with this option we have an effective size of the key of 168. But these option is weak by “meet in the middle attack” and however the effective size of the key for a “brute force attack” is 168, if we use the “meet in the middle attack” the effective size of the key is 112[30].
The best attack known to this option would need so many plain texts, steps, single DES encryptions and memory that it is impossible to use at the practice[29]
2. Key1 = Key3 with Key2 different : with this option we have an effective size of key of 112 bits (56bits x 2), because we have two equal keys(Key1=Key3). But, this option is susceptible to certain chosen-plaintext or known-plaintext attacks[31, 32]
3. Key1 = Key2 = Key3 : this option is the equal that use DES, because the encryption with the first key is annulled by the decryption of the second, so the result is the encrypted text with the third key. Because this, this option only have an effective size of key of 56 bits. It is not recommended use this option because this option spends more time and resources than DES and has the same security[30]

3.2.3 Practice analysis

In our practical work, we have measured the time consumption during the encryption and decryption. Since the OS eventually expropriates the CPU, we have done rounds of 100.000 and 1.000.000 repetitions and then, we show the average of them.

The measured times are the following :

Encryption :			
Message Size	Encrypted Message Size	100.000 repetitions	1.000.000 repetitions
56 bits	64 bits	2562.17 ns	2544.78 ns
64 bits	128 bits	4182.54 ns	3963.33 ns
128 bits	192 bits	5441.93 ns	5435.20 ns
256 bits	320 bits	8375.24 ns	8373.70 ns
512 bits	576 bits	14243.30 ns	14229.06 ns
1024 bits	1088 bits	25899.92 ns	25878.54 ns

Decryption :		
Message Size	100.000 repetitions	1.000.000 repetitions
56 bits	2166.53 ns	2165.80 ns
64 bits	3587.26 ns	3580.78 ns
128 bits	4993.65 ns	4984.07 ns
256 bits	7782.37 ns	7777.64 ns
512 bits	13348.89 ns	13336.77 ns
1024 bits	24464.36 ns	24431.83 ns

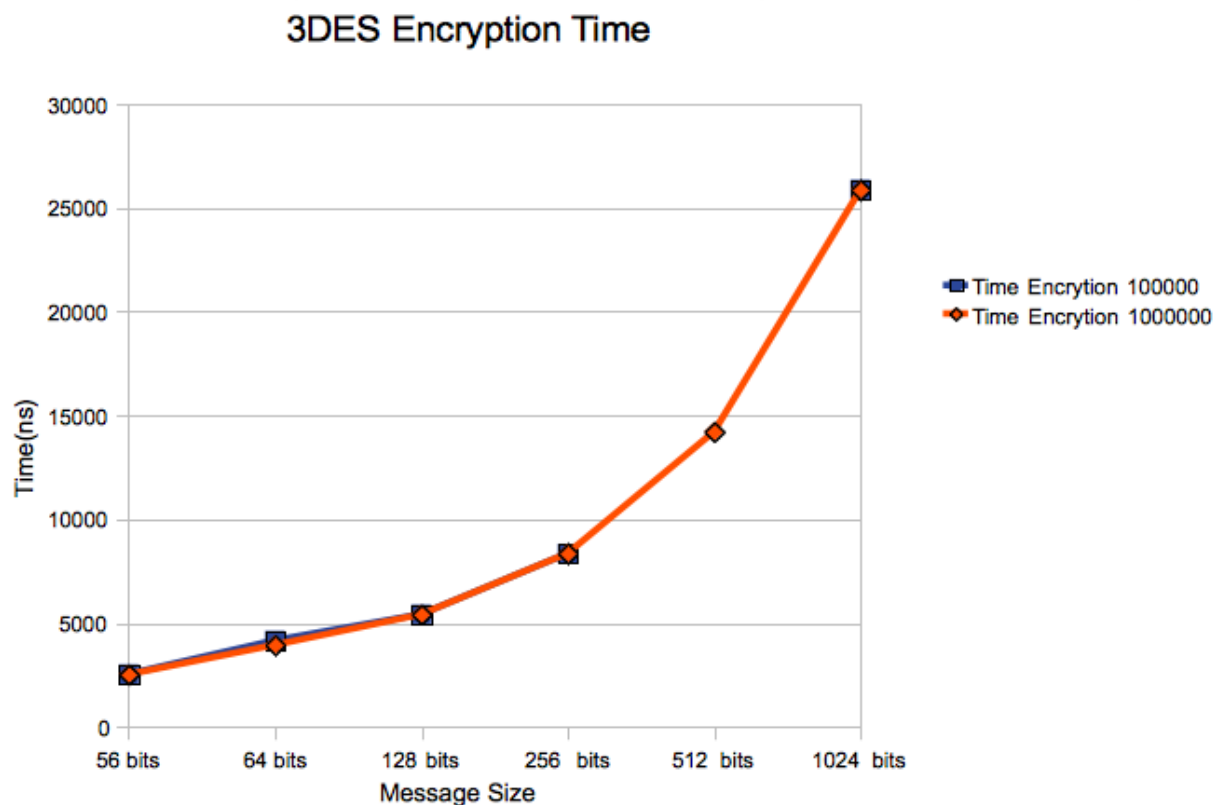


Figure 3.2.3: 3DES Encryption

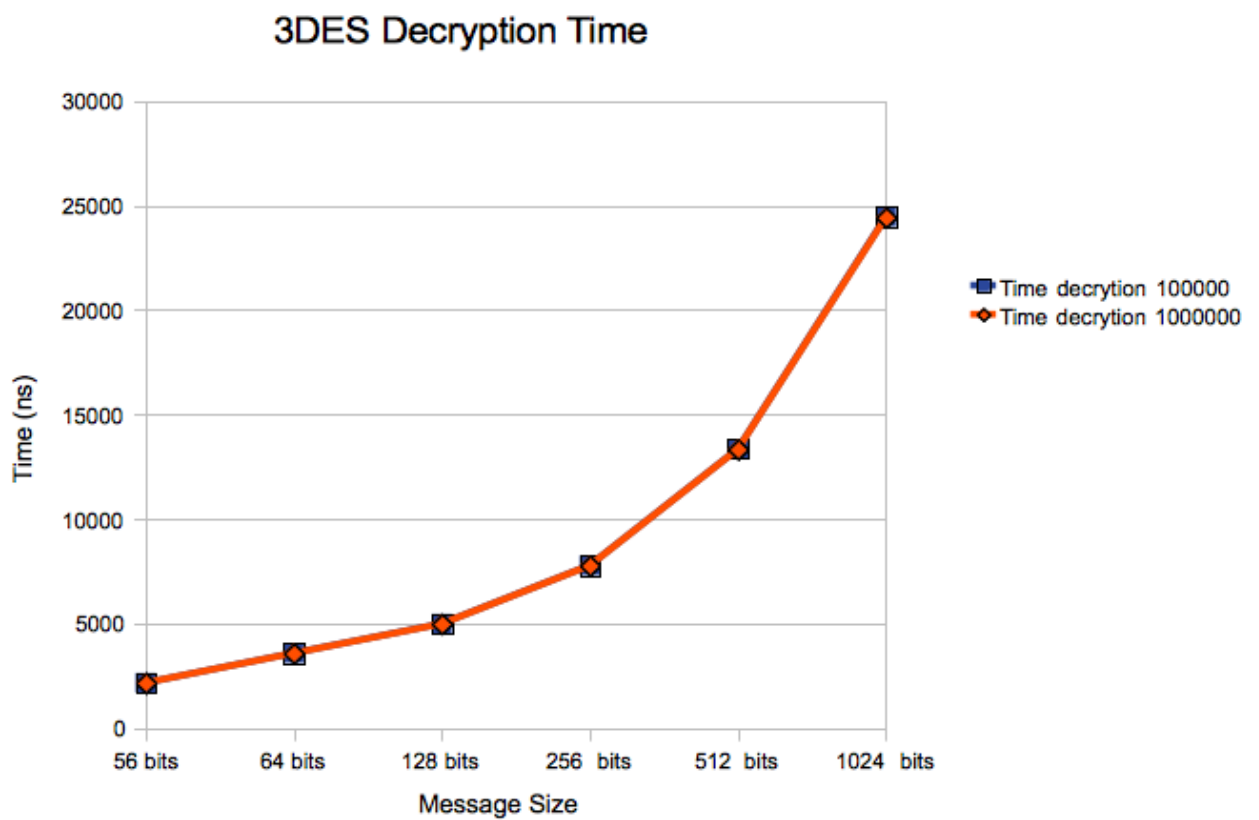


Figure 3.2.4: 3DES Decryption

3.3 Advanced Encryption Standard (AES)

The actual *AES* algorithm was originally published as *Rijndael* in 1998. *Rijndael* was created by Joan Daemen and Vincent Rijmen and it was the winner of the AES selection process, and announced by the *National Institute of Standards and Technology (NIST)* on November 2001 as U.S. FIPS PUB 197 (FIPS 197)[9].

3.3.1 Theoretical description

AES is based on a *Substitution permutation network* instead of a *Feistel Network* as *DES* [3.1] and *Blowfish* [3.4]. It supports a block length of 128, 192 or 256 bits. There's another variable in this algorithm, called *State*[8].

The *State* is a 4-row table of byte values. If the block size is 128 bit, the *State* will have 4 columns. If it is a 192 bits long, there will be 6 columns on the *State* and 8 for the 256 bit length block. The table is filled by columns instead of rows. An example is shown in the following table for a block length of 128 bit (16 bytes)

State example :

Byte 1	Byte 5	Byte 9	Byte 13
Byte 2	Byte 6	Byte 10	Byte 14
Byte 3	Byte 7	Byte 11	Byte 15
Byte 4	Byte 8	Byte 12	Byte 16

The algorithm encrypts the data in 3 several identical rounds. On each round another subkey is included. Before the bytes are processed, they are entered in the *State*.

The steps are : [8]

1. Initial Round

- AddRoundKey

2. Rounds

- ByteSub
- ShiftRow
- MixColumn
- AddRoundKey

3. Last Round

- ByteSub
- ShiftRow
- AddRoundKey

Description of each function :

- AddRoundKey : This function prepares the cipher for the next round, that's why on the initial round it is the only function and then on each round it's done at the end.

AddRoundKey adds the *State* to a *round-key* which is derived from the key. The *round-key* is a byte table with four rows and it has the same column amount as the *State*. Therefore, the column amount depends on the block length.

- ByteSub : As is shown before, it is the start of the encryption on each round.
 1. Each byte is considered as an element of a *Galois field* [10] of size 2^8 or $\text{GF}(2^8)$. Then the byte is replaced by the inverse element of the multiplication in $\text{GF}(2^8)$.
 2. The byte to encrypt is then multiplied by an 8×8 matrix M and a 8 bit value is added. Let the byte be b and the value added b^0 .
 The 8×8 matrix is build based on b^0 .
 The first row is b^0 itself.
 The 2nd row is b^0 shifted right 1 position.
 The i -row is b^0 shifted i -times. As a result we obtain $b = M * b + b^0$

Both operations acts as a S-box like in *DES*.

- ShiftRow : The second operation on each round.
 The four rows on the *State* are rotated to the right a variable number of columns. This is set up depending on the block lenght. Therefore, ShiftRow only mixes the contents of the *State*.
- MixColumn :
 This function mixes the data according to columns right after ShiftRow did it by rows. In this case, the mixing is a bit more complicated. MixColumn takes the *State* bytes as elements of $\text{GF}(2^8)$. They are also treated as a vector and they are multiplied by a matrix.

3.3.2 Algorithm's security

AES has 10 rounds for 128-bit keys, 12 rounds for 192-bit keys, and 14 rounds for 256-bit keys. By 2006, the best known attacks were on 7 rounds for 128-bit keys, 8 rounds for 192-bit keys, and 9 rounds for 256-bit keys[21].

Until 2009 only *side-channel attacks* were successful. They do not attack the underlying cipher but they attack implementations of the cipher on systems where data is leaked. OpenSSL's AES encryption was broken by a cache-timing attack in 2005[22] in a custom server designed to give out as much timing information as possible and the attack required 200 million chosen plaintexts[23].

There is a known-key distinguishing attack[25] against a reduced 8-round version of AES-128 with a computation complexity of 2^{48} .

There is one attack against AES-256 [24] that uses only two related keys and 2^{39} time to recover the complete 256-bit key of a 9-round version, 2^{45} for a 10-round version and 2^{70} for a 11-round version. Since 256-bit AES uses 14 rounds, these attacks aren't effective against full AES.

Differential Fault Analysis is an attack used on some hardware implementations and allows recovery of key with complexity of 2^{32} [26].

3.3.3 Practice analysis

In our practical work, we have measured the time consumption during the encryption and decryption. Since the OS eventually expropriates the CPU, we have done rounds of 100.000 and 1.000.000 repetitions and then, we show the average of them.

The measured times are the following :

128 bit key lenght:

		Encryption :		Decryption :	
Original Size	Encrypted Size	100k rep.	1.000k rep.	100k rep.	1.000k rep
56 bits	128	909.14 ns	913.45 ns	764.15 ns	750.13 ns
64 bits	128	937.79 ns	929.12 ns	766.01 ns	759.62 ns
128 bits	256	1245.29 ns	1241.44 ns	1019.60 ns	1015.10 ns
256 bits	384	1687.69 ns	1632.06 ns	1264.51 ns	1258.06 ns
512 bits	640	2390.09 ns	2336.23 ns	1773.03 ns	1760.07 ns
1024 bits	1152	3763.40 ns	3760.54 ns	2762.10 ns	2759.10 ns

192 bit key lenght:

		Encryption :		Decryption :	
Original Size	Encrypted Size	100k rep.	1.000k rep.	100k rep.	1.000k rep
56 bits	128	1014.40 ns	1010.24 ns	866.57 ns	864.56 ns
64 bits	128	1048.35 ns	1037.17 ns	874.26 ns	864.03 ns
128 bits	256	1383.40 ns	1379.44 ns	1163.71 ns	1160.38 ns
256 bits	384	1814.44 ns	1799.63 ns	1455.37 ns	1452.25 ns
512 bits	640	2600.98 ns	2589.50 ns	2052.78 ns	2050.78 ns
1024 bits	1152	4224.24 ns	4165.89 ns	3214.89 ns	3243.77 ns

256 bit key lenght:

		Encryption :		Decryption :	
Original Size	Encrypted Size	100k rep.	1.000k rep.	100k rep.	1.000k rep
56 bits	128	1083.70 ns	1080.14 ns	954.67 ns	949.26 ns
64 bits	128	1111.23 ns	1107.73 ns	956.11 ns	953.41 ns
128 bits	256	1483.79 ns	1483.20 ns	1300.00 ns	1297.13 ns
256 bits	384	1911.09 ns	1905.10 ns	1609.16 ns	1609.04 ns
512 bits	640	2810.22 ns	2789.63 ns	2316.11 ns	2296.10 ns
1024 bits	1152	4504.77 ns	4530.69 ns	3601.51 ns	3597.68 ns

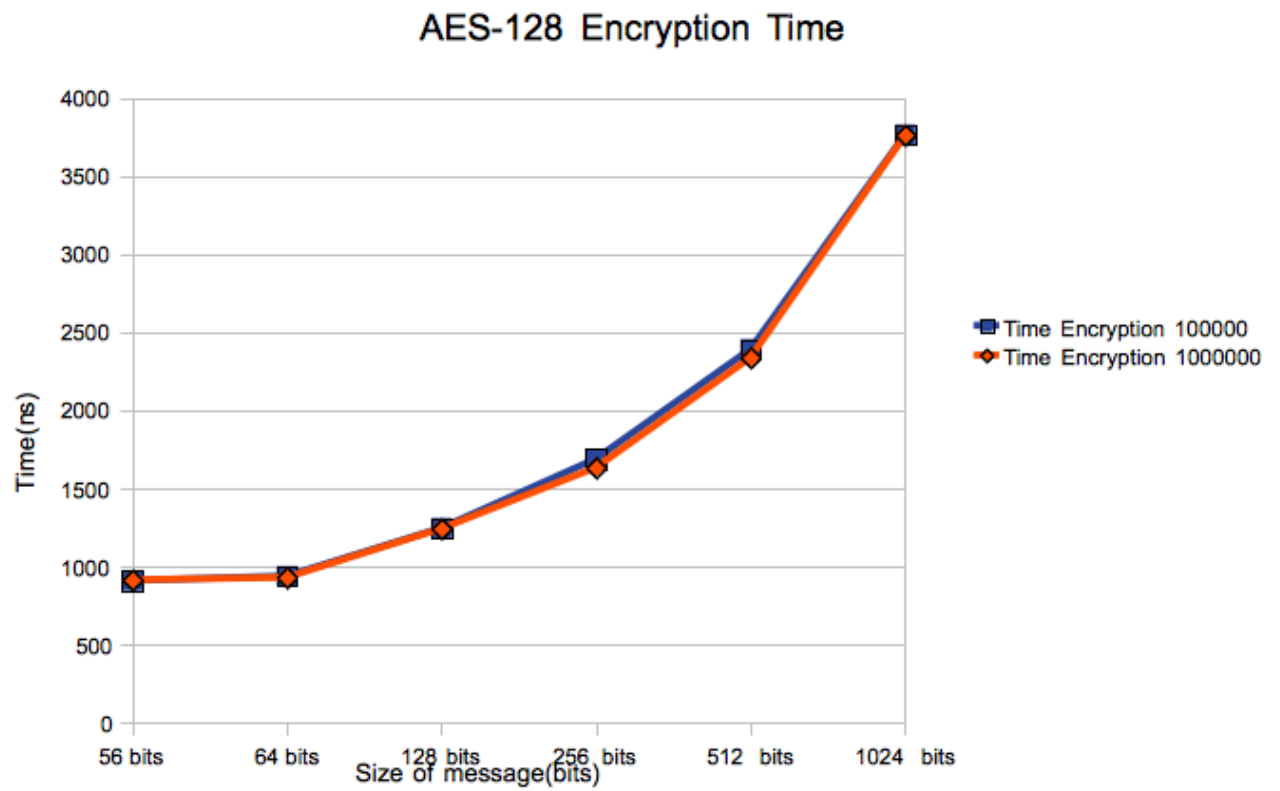


Figure 3.3.1: AES128 Encryption

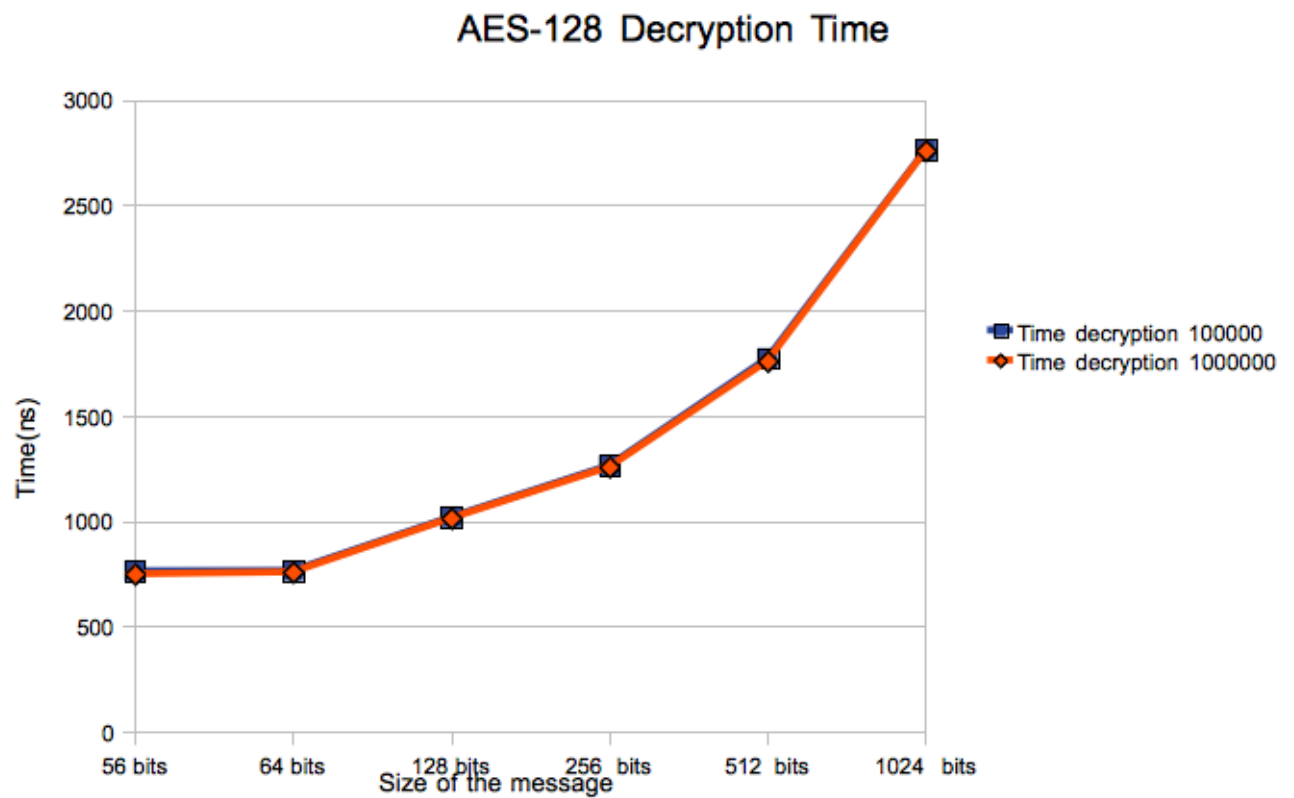


Figure 3.3.2: AES128 Decryption

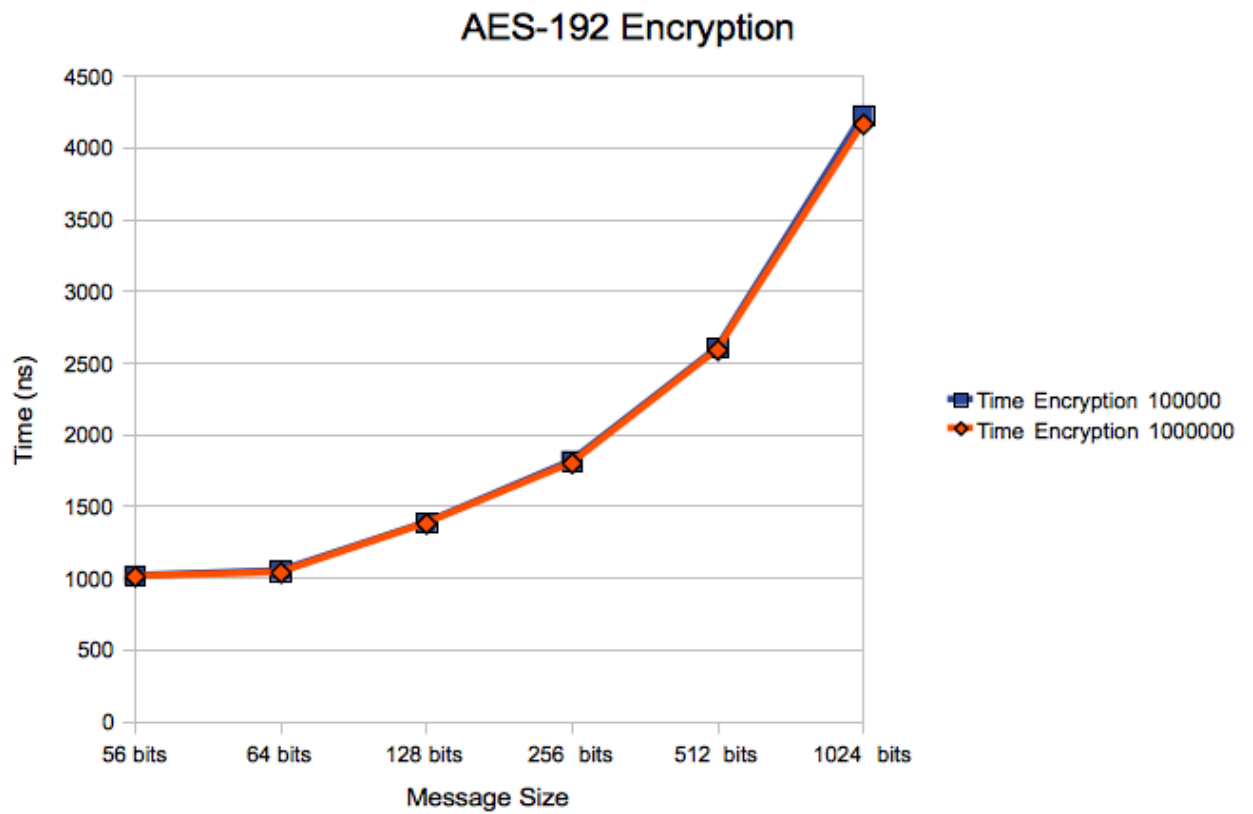


Figure 3.3.3: AES192 Encryption

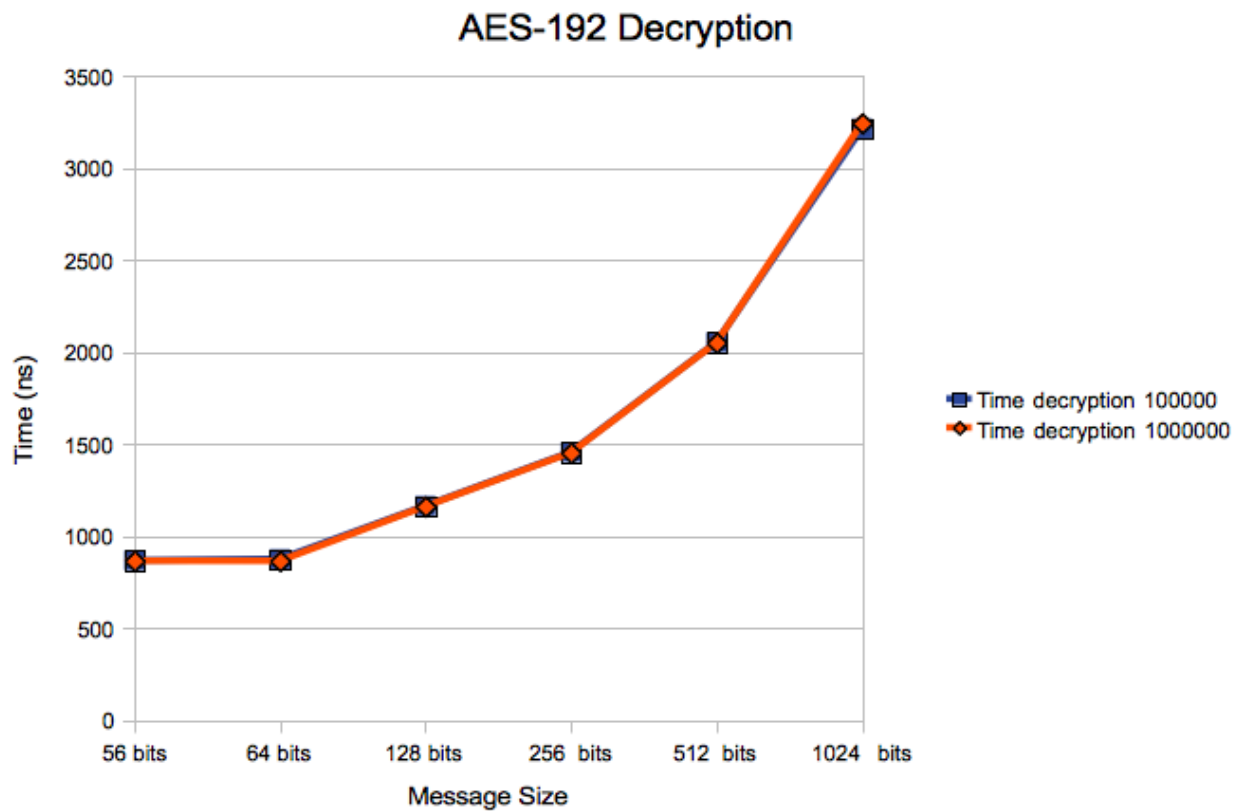


Figure 3.3.4: AES192 Decryption

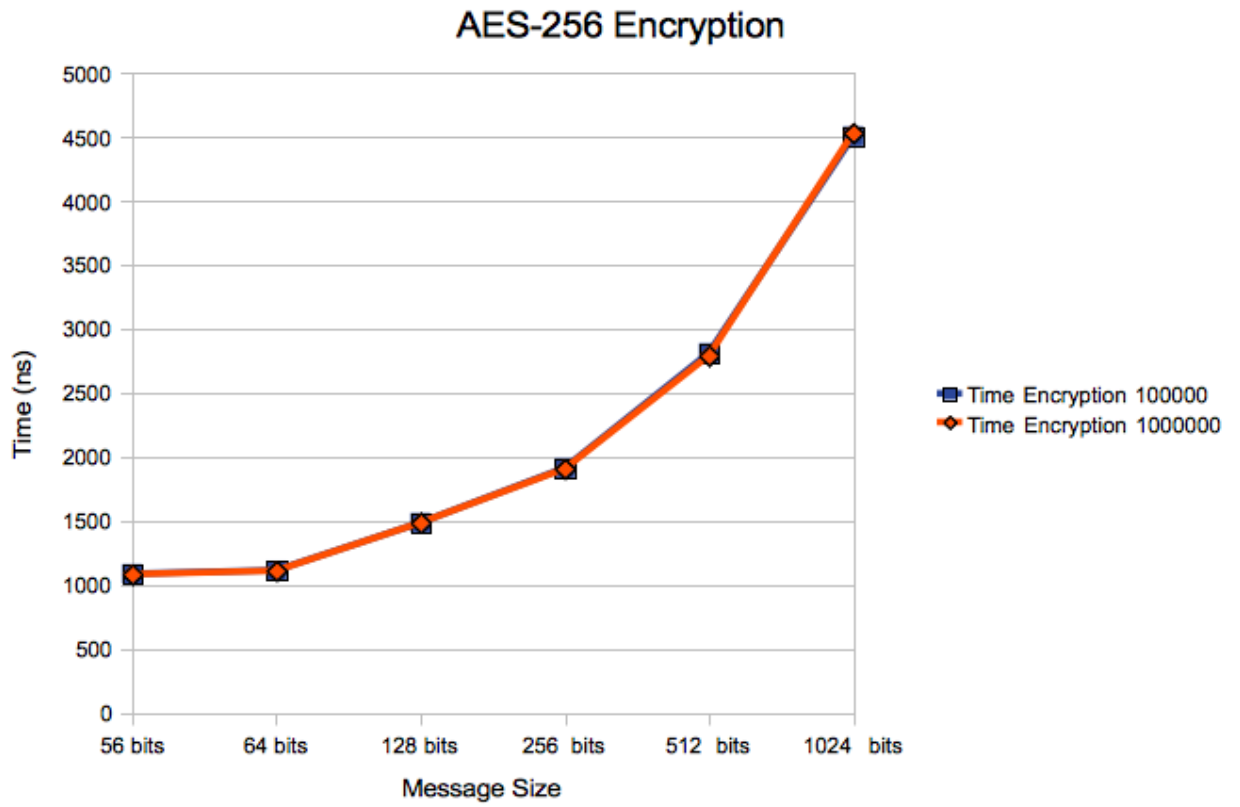


Figure 3.3.5: AES256 Encryption

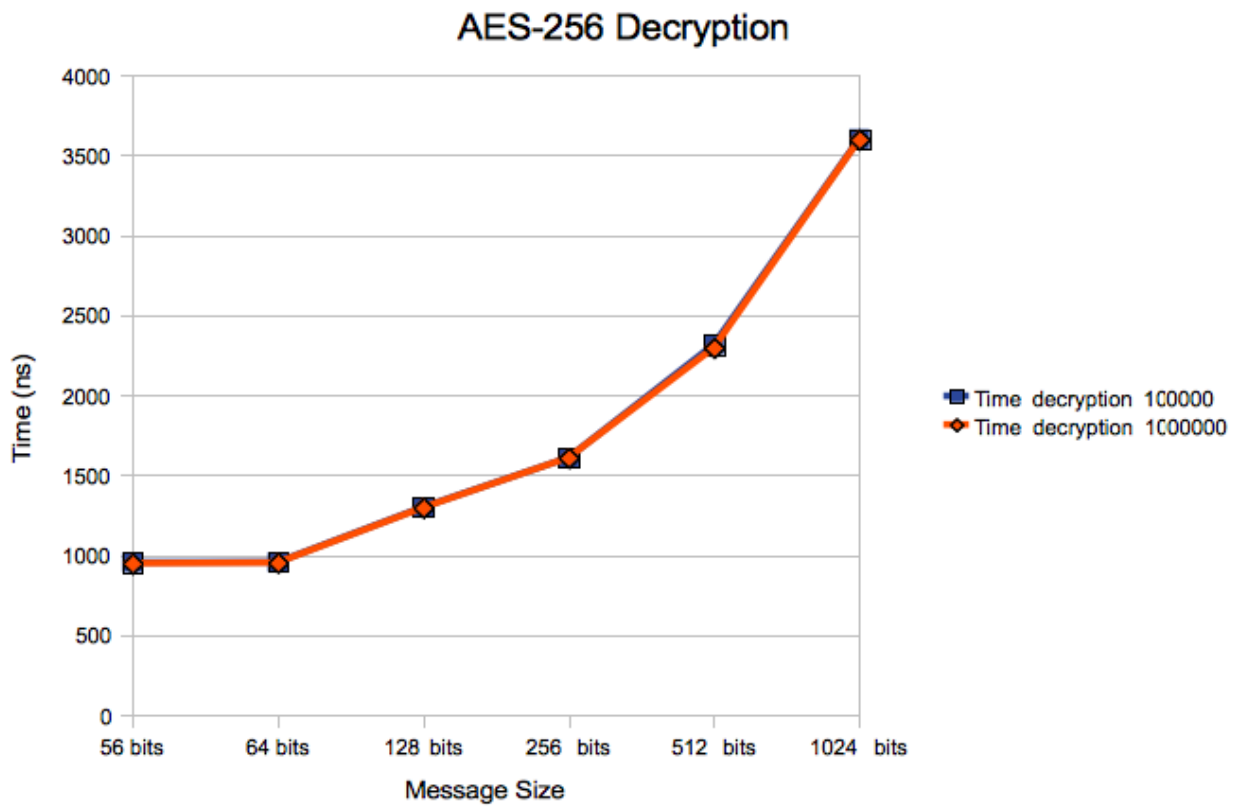


Figure 3.3.6: AES256 Decryption

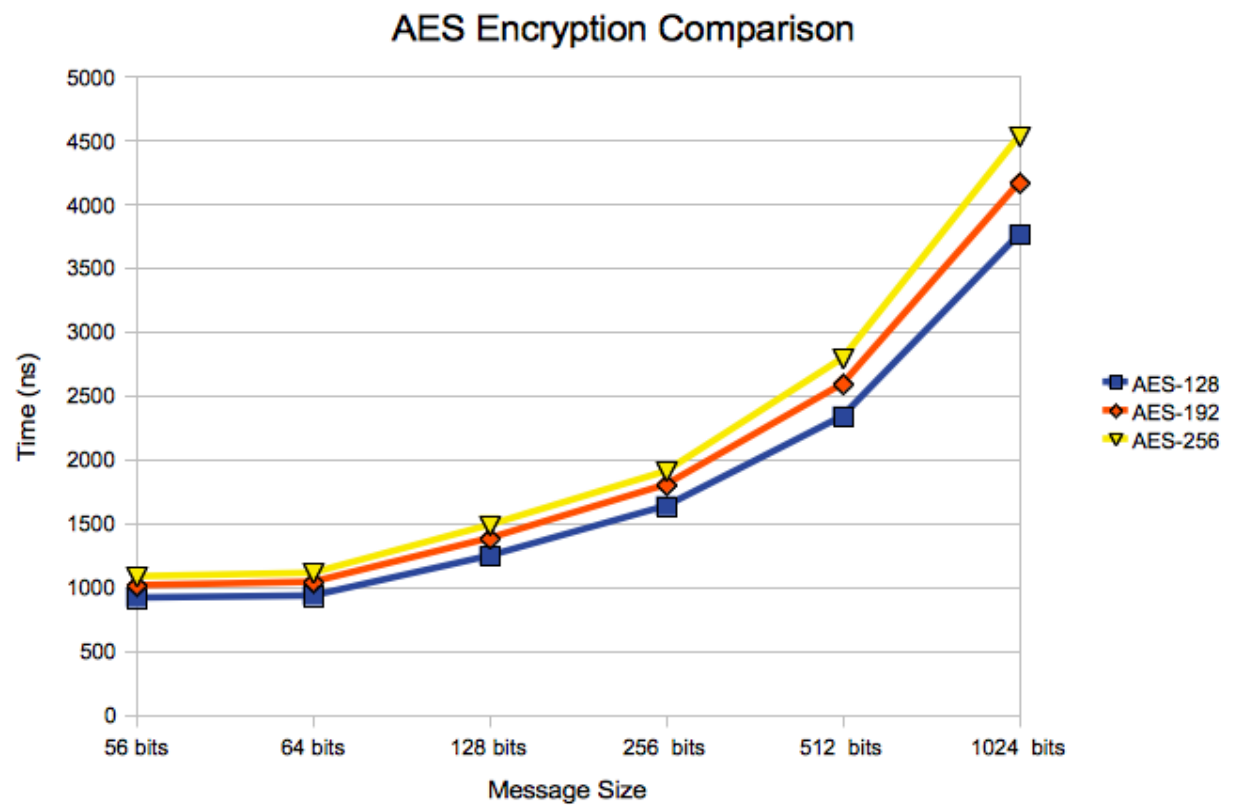


Figure 3.3.7: AES Encryption comparison

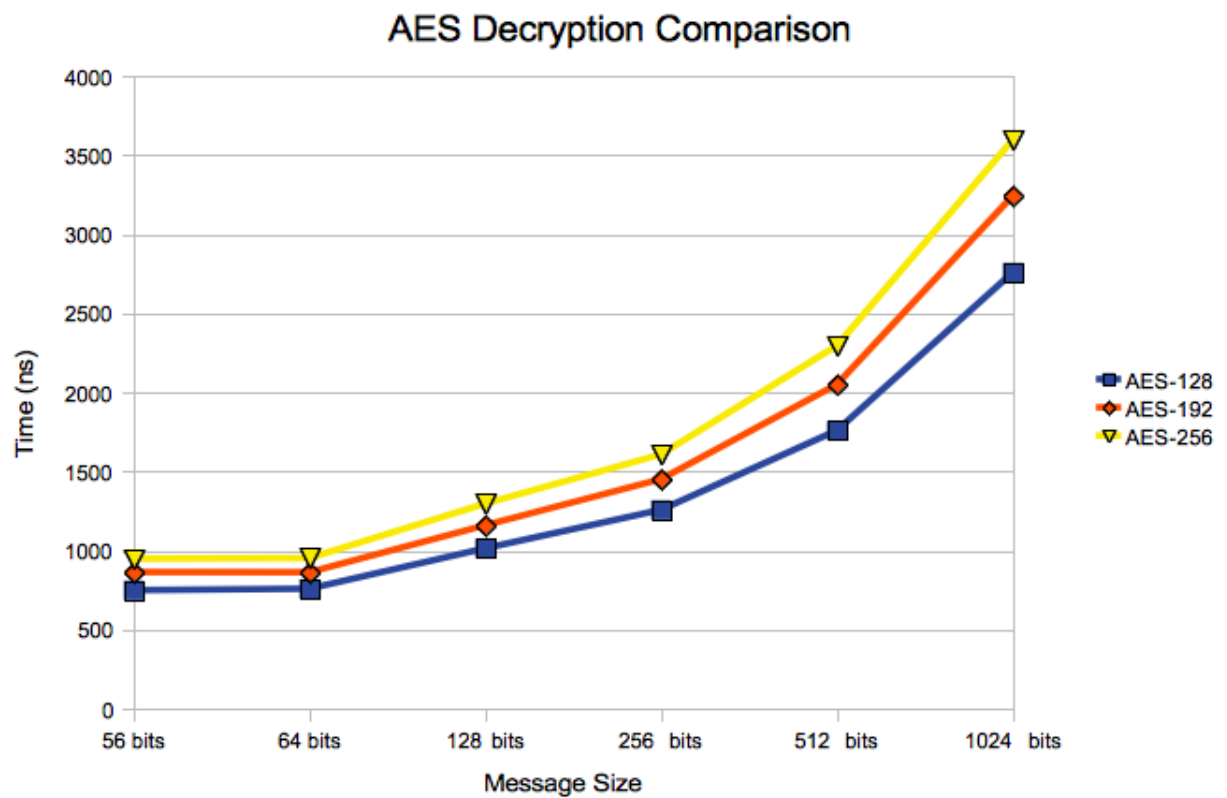


Figure 3.3.8: AES Decryption comparison

3.4 Blowfish

Presented in 1994 by Bruce Schneier and presented as an unpatented algorithm. It is added to Linux kernel since v2.5.47.

3.4.1 Theoretical description

Blowfish is a secret-key block cipher which uses 64bits blocks. *Blowfish* applies a simple encryption function 16 times via a Feistel network as it is shown in Figure 3.4.1, where each line represents 32 bits.

It is capable of managing keys from 8 to 448 bits in steps of 8 bits (Figure 3.4.2).

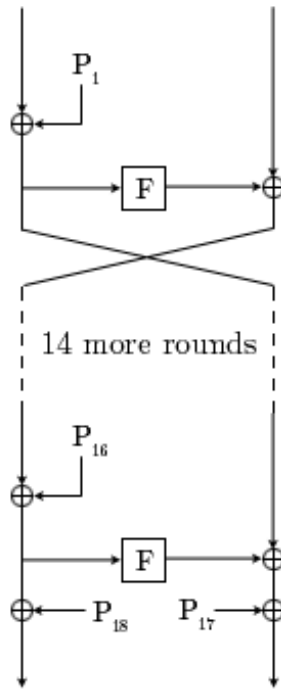


Figure 3.4.1: Blowfish : Feistel network[36]

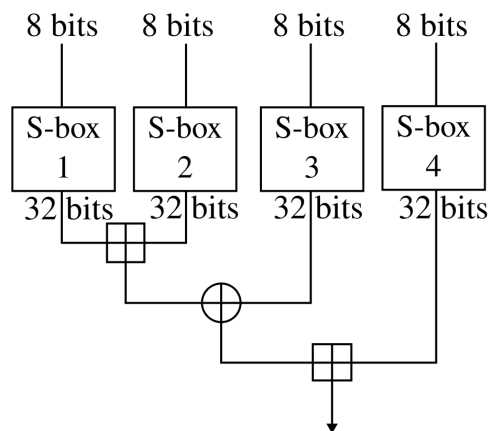


Figure 3.4.2: Blowfish : Round Function[41]

On each step, it applies a key-dependent permutation and a key&data substitution. To achieve this, the algorithm keeps two subkey arrays : [11]

1. 18-entry P-array ($P_1 \dots P_{18}$) :
Used from $P_1 \dots P_{18}$ for encryption and from $P_{18} \dots P_1$ for decryption. One P_i is used on one iteration. Each P-array is 32-bit.
2. Four 256-entry S-boxes :
They accept 8-bit inputs and they produce 32-bit outputs.

Since *Blowfish* uses large keys, they have to be splitted in a large number of subkeys. Those keys must be computed before any data encryption/decryption.

The sub-key generation has the following steps :[11]

1. P-array is initialized and then the 4 S-boxes in order with a fixed string. That string consists of the hexadecimal digits of pi
2. XOR P_1 with the first 32 bits of the key, XOR P_2 with the second 32-bit (and so on).
3. Encrypt a 64-bit all-zero string with the *Blowfish* algorithm using the previous subkeys.
4. Replace P_1 and P_2 with the output of step 3
5. Encrypt the output of step 3 using the *Blowfish* algorithm with the modified subkeys.
6. Replace P_3 and P_4 with the output of step 5
7. Continue the process with the output of the continuously-changing algorithm

As a consequence of all this steps, 521 iterations are required to generate all required subkeys. Because the P-array is 576 bits long, and the key bytes are XORed through all these 576 bits during the initialization, many implementations support key sizes up to 576 bits. It is limited to 448 bits to ensure that every key of every subkey depends on every bit of the key [12] (the last 4 P_i does not affect every bit of the ciphertext).

Due to its initial sub-key calculus, a slow initialization of the cipher (when the key is changed) is granted. Thanks to the enormous key, a brute-force attack protection is granted.

3.4.2 Algorithm's security

Concerning Blowfish security, there is no effective cryptanalysis on the full-round version published yet.

On the other hand, there is a know-plaintext attack requiring 2^{8r+1} known plaintext to break, where the r is the number of rounds. There are some *weak keys*[15] that can be detected with only 2^{4r+1} known plaintexts. But this kind of attack can be only used if we know the S-boxes content.

There is another attack which can break four rounds but no more[16]. There's no known way to break the full 16 rounds, only a brute-force search.

3.4.3 Practice analysis

32 bit key lenght:

		Encryption :		Decryption :	
Original Size	Encrypted Size	100k rep.	1.000k rep.	100k rep.	1.000k rep
56 bits	128	578.12 ns	573.03 ns	374.54 ns	371.20 ns
64 bits	128	812.69 ns	817.00 ns	533.93 ns	531.97 ns
128 bits	192	1056.57 ns	1044.01 ns	638.07 ns	650.90 ns
256 bits	320	1523.61 ns	1529.79 ns	936.96 ns	937.27 ns
512 bits	576	2491.36v	2469.62 ns	1468.91 ns	1457.26 ns
1024 bits	1088	4294.95 ns	4287.88 ns	2502.35 ns	2500.97 ns

64 bit key lenght:

		Encryption :		Decryption :	
Original Size	Encrypted Size	100k rep.	1.000k rep.	100k rep.	1.000k rep
56 bits	128	638.05 ns	626.89 ns	379.04 ns	377.99 ns
64 bits	128	813.78 ns	805.92 ns	534.88 ns	535.25 ns
128 bits	192	1050.05 ns	1052.49 ns	717.13 ns	699.67 ns
256 bits	320	1531.73 ns	1543.21 ns	944.06 ns	934.53 ns
512 bits	576	2474.97 ns	2476.99 ns	1460.84 ns	1459.55 ns
1024 bits	1088	4317.02 ns	4309.99 ns	2511.41 ns	2512.13 ns

128 bit key lenght:

		Encryption :		Decryption :	
Original Size	Encrypted Size	100k rep.	1.000k rep.	100k rep.	1.000k rep
56 bits	128	583.01 ns	588.40 ns	378.29 ns	379.31 ns
64 bits	128	796.64 ns	786.20 ns	535.61 ns	532.44 ns
128 bits	192	1088.29 ns	1076.30 ns	668.87 ns	699.74 ns
256 bits	320	1535.26 ns	1543.91 ns	936.61 ns	935.42 ns
512 bits	576	2481.11 ns	2479.69 ns	1458.87 ns	1457.20 ns
1024 bits	1088	4324.36 ns	4313.45 ns	2505.78 ns	2502.95 ns

256 bit key lenght:

		Encryption :		Decryption :	
Original Size	Encrypted Size	100k rep.	1.000k rep.	100k rep.	1.000k rep
56 bits	128	581.88 ns	583.33 ns	379.64 ns	391.27 ns
64 bits	128	793.26 ns	792.99 ns	534.61 ns	532.44 ns
128 bits	192	1086.70 ns	1085.97 ns	671.98 ns	668.07 ns
256 bits	320	1569.87 ns	1591.83 ns	936.32 ns	933.19 ns
512 bits	576	2600.36 ns	2599.92 ns	1461.28 ns	1461.34 ns
1024 bits	1088	4329.77 ns	4329.62 ns	2526.11 ns	2522.83 ns

448 bit key lenght:

		Encryption :		Decryption :	
Original Size	Encrypted Size	100k rep.	1.000k rep.	100k rep.	1.000k rep
56 bits	128	614.38 ns	615.53 ns	379.72 ns	378.68 ns
64 bits	128	801.79 ns	801.43 ns	535.41 ns	533.02 ns
128 bits	192	1072.97 ns	1070.74 ns	674.22 ns	674.71 ns
256 bits	320	1539.48 ns	1532.10 ns	944.61 ns	944.29 ns
512 bits	576	2462.76 ns	2450.61 ns	1485.02 ns	1482.09 ns
1024 bits	1088	4349.34 ns	4426.02 ns	2610.67 ns	2558.28 ns

In Figure 3.4.3 we can see how the encryption time increases with the message size.

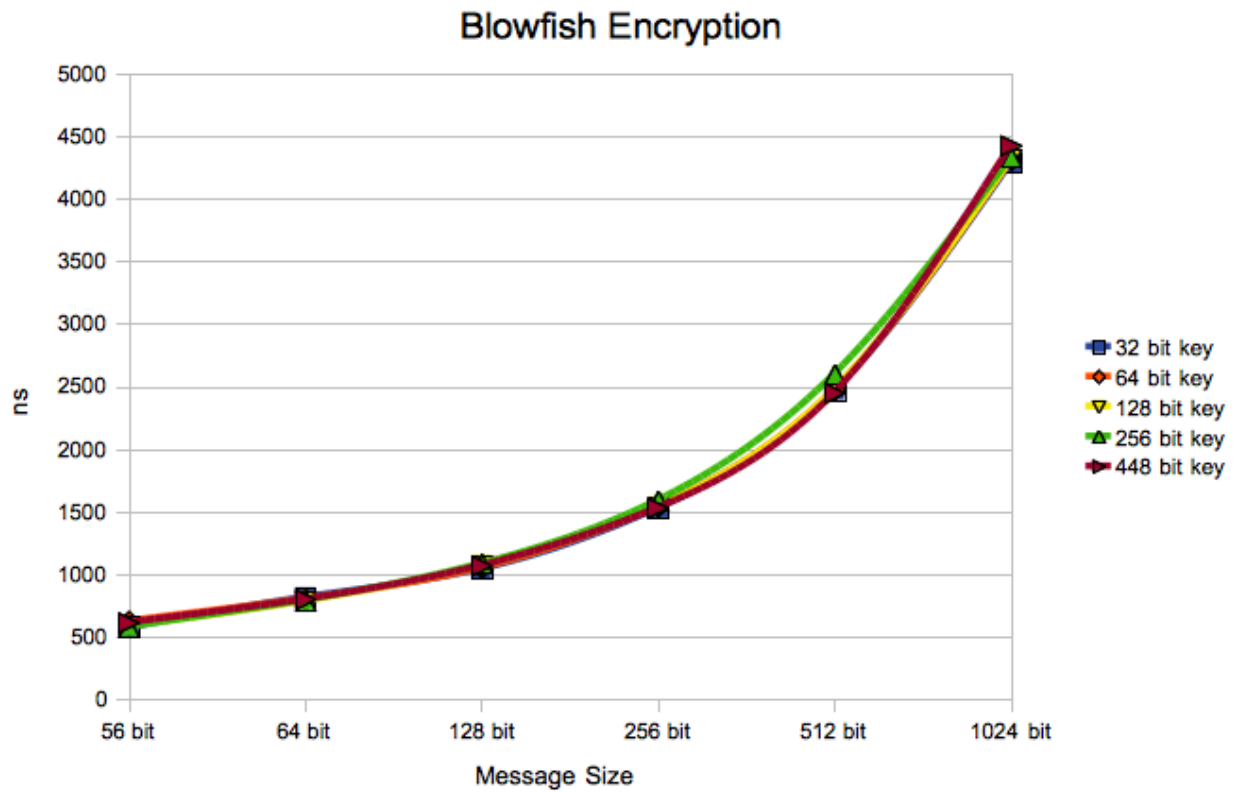


Figure 3.4.3: Blowfish : Encryption time vs message size

As we can see on the graph, the encryption time only is increased by the message size, not by the key length. This leads to a better security without affecting to the performance. There is no reason to use smaller key lengths. The same happens with the decryption, as is show in Figure 3.4.4.

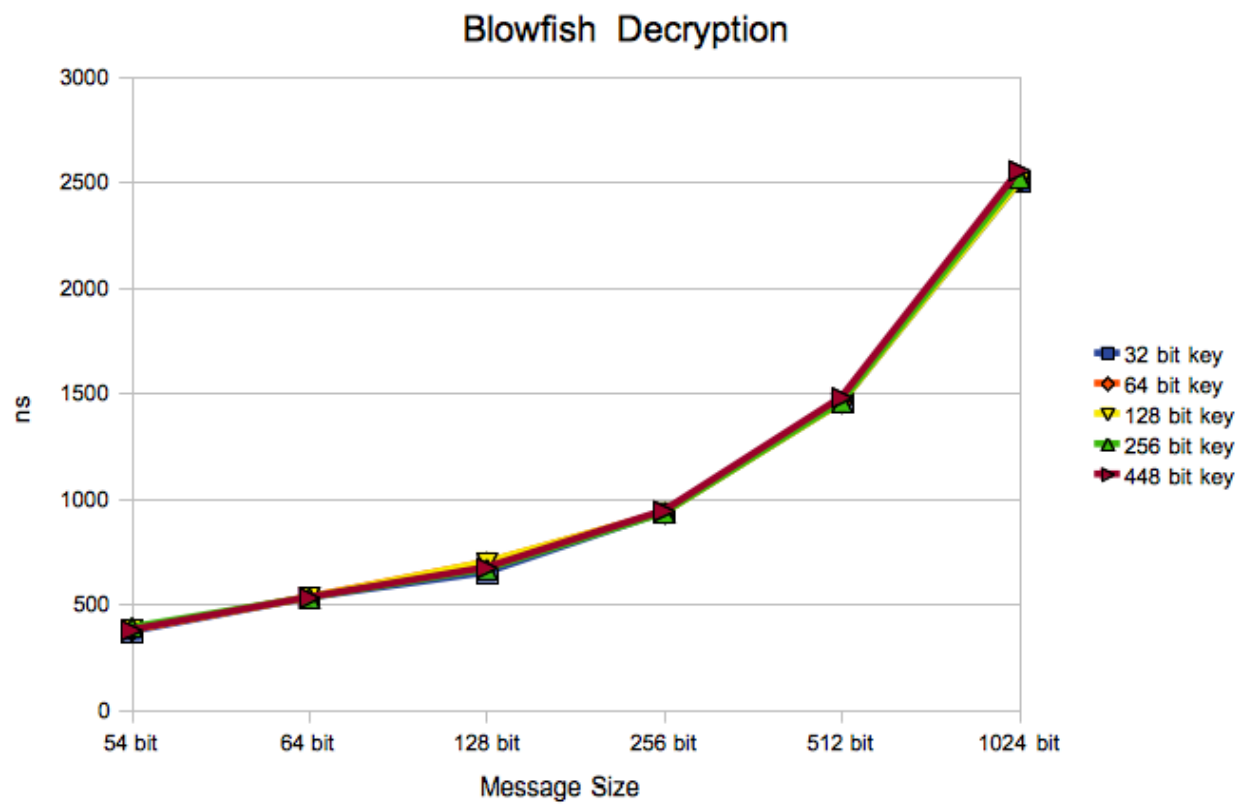


Figure 3.4.4: Blowfish : Encryption time vs message size

3.5 Serpent

Serpent is another symmetric key block cipher designed by *Ross Anderson*, *Eli Biham* and *Lars Knudsen* in 1998.[13] This algorithm is public domain, like the blowfish, and can be used by anyone.

3.5.1 Theoretical description

Serpent uses block sizes of 128 bits and the keys can be 128, 192 or 256-bit long.

Serpent is a 32-round substitution-permutation network operating on a block of four 32-bit words. Each round applies one of eight 4-bit to 4-bit S-boxes 32 times in parallel.

All operations can be executed in parallel using 32 1-bit slices. By that, maximizes parallelism and allows use of the cryptanalysis work performed on DES. It is much faster than DES (as we will be able to see in Section 3.5.3.)

Serpent is designed to allow a single block to be encrypted efficiently by bitslicing. It achieves a high performance with a very efficient use of parallelism.

The cipher consists of :[13]

- Initial permutation IP
- 32 rounds, each of them do a key mixing operation (Figure 3.5.1), a pass through S-boxes and a linear transformation. On the last round, the linear transformation is replaced by and additional key mixing.
- A final permutation FP

Initial and final permutations are used to simplify an optimized implementation of the cipher, to improve its computational efficiency.

IP is applied to the plaintext P giving B_0 (which is the input to the 1st round).

The rounds are numbered from 0 to 31. The output of round i is B_{i+1} and the last output is denoted by B_{32}

Then the final permutation FP is applied to give the ciphered text (C).

Each round function R_i ($i \in \{0, \dots, 31\}$) uses only a single replicated S-box (see 3.1.4).

R_0 uses S_0 , 32 copies of which are applied in parallel.

The first copy of S_0 takes bits 0,1,2 and 3 of $B_0 \oplus K_0$ (being K_0, \dots, K_{32} subkeys) as its input and returns as output the first four bits of an intermediate vector.

The 32 rounds use 8 different S-boxes each of which maps four input bits to four output bits. Each S-box is used 4 times (in different rounds) and each of them is used 32 times in parallel.

The formal description is as follows, beign S_i the application of the S-box $S_{i \bmod 8}$ 32 times in parallel and L the linear transformation.

$$\begin{aligned} B_0 &:= IP(P) \\ B_{i+1} &:= R_i(B_i) \\ C &:= FP(B_{32}) \end{aligned}$$

Where :

$$\begin{aligned} R_i(X) &= L(S_i(X \oplus K_i)) \text{ having } i = 0, \dots, 30 \\ R_i(X) &= S_i(X \oplus K_i) \oplus K_{32} \text{ with } i = 1 \end{aligned}$$

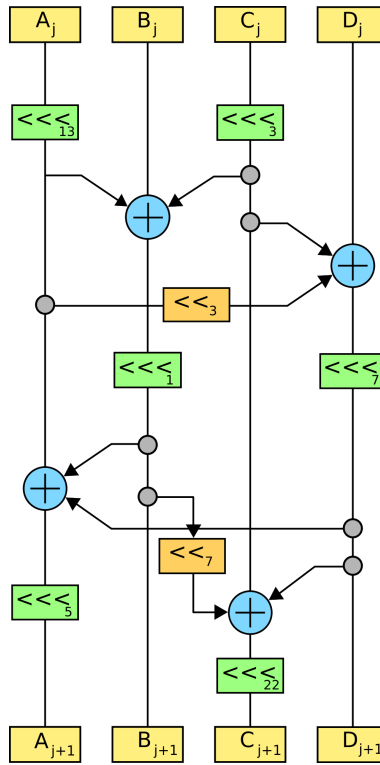


Figure 3.5.1: Key mixing on Serpent[39] where \lll is a rotation and \ll is a shift

S-boxes:

This algorithm uses S-boxes as DES and Blowfish but in a different way. S-boxes are generated using a matrix with 32 arrays each with 16 entries. First the matrix is initialised with the 32 rows of the DES S-boxes. They are transformed by swapping the entries. The entries in r -th array depends on the value of the entries in the $(r+1)$ th and on an initial string representing a key. This is repeated until the 8 S-boxes have been generated.

Pseudo-code:[13]

```

index := 0
repeat
    currentsbbox := index modulo 32;
    for i:=0 to 15 do
        j := sbbox[(currentsbbox+1) modulo 32][serpent[i]];
        swapentries (sbbox[currentsbbox][i],sbbox[currentsbbox][j]);
        if sbbox[currentsbbox][.] has the desired properties, save it;
        index := index + 1;
until 8 S-boxes have been generated\index{S-box}

```

Decryption is different from encryption in that the inverse of the S-boxes must be used in the reverse order, as well as the inverse linear transformation and reverse order of the subkeys.

3.5.2 Algorithm's security

There is a *meet-in-the-middle* attack against 6 of 32 rounds of Serpent [27] and another attack against 9 of 32 rounds[27] called *amplified boomerang attack*. For the *meet-in-the-middle* attack, 2^{83} chosen plaintexts, 2^{20} bytes of memory and 2^{90} trial encryptions are being used to attack six-round Serpent. The *boomerang amplified* for 9-round Serpent uses 2^{110} chosen plaintexts, 2^{212} bytes of memory and approximately 2^{252} trial encryptions.

Since those attacks can only reach up to 9 rounds, the full 32-round Serpent is secure.

3.5.3 Practice analysis

In our practical work, we have measured the time consumption during the encryption and decryption. Since the OS eventually expropriates the CPU, we have done rounds of 100.000 and 1.000.000 repetitions and then, we show the average of them.

The measured times are the following :

		Encryption :		Decryption :	
Original Size	Encrypted Size	100k rep.	1.000k rep.	100k rep.	1.000k rep
56 bits	128	1443.96 ns	1450.55 ns	1123.60 ns	1112.98 ns
64 bits	128	1412.06 ns	1421.35 ns	1118.64 ns	1148.34 ns
128 bits	256	1958.28 ns	1946.16 ns	1688.4 ns	1701.60 ns
256 bits	384	2521.24 ns	2593.33 ns	2171.71 ns	2165.80 ns
512 bits	640	3804.05 ns	3733.88 ns	3246.44 ns	3195.65 ns
1024 bits	1152	6070.26 ns	6061.87 ns	5249.81 ns	5223.51 ns

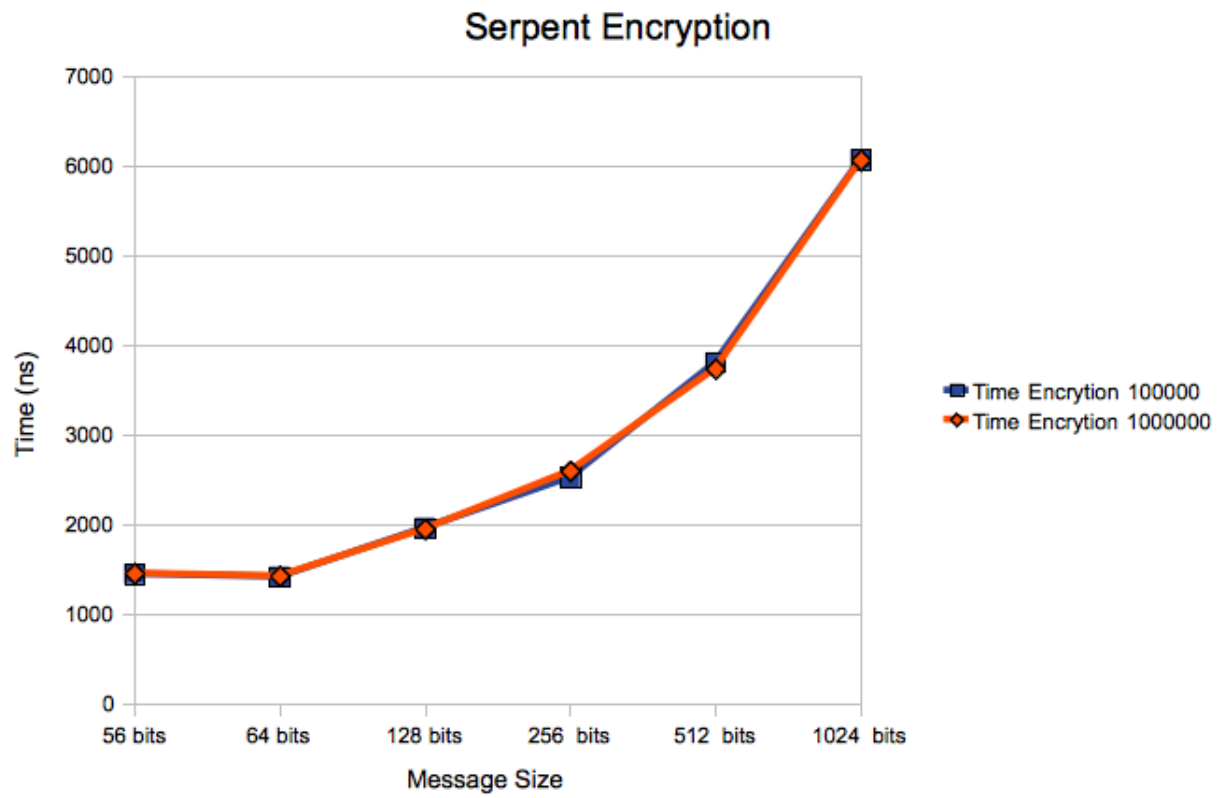


Figure 3.5.2: Serpent Encryption

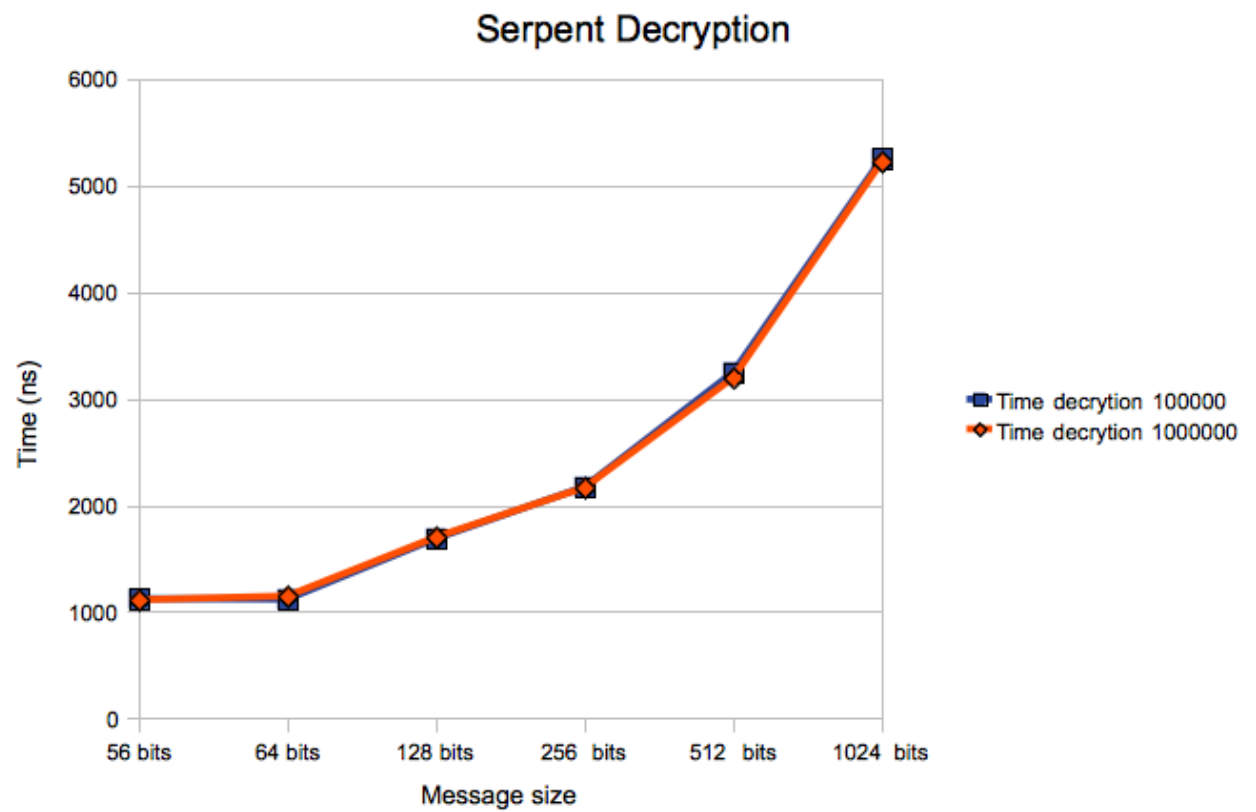


Figure 3.5.3: Serpent Decryption

3.6 Rivest Shamir Adleman (RSA)

The RSA cryptosystem[2] is the most widely known digital signature system and the simplest to understand. It was invented by R. Rivest, A. Shamir, and L. Adleman in 1977 but the basic technique was invented in 1973 [3].

3.6.1 Theoretical description

RSA is a two key algorithm, one key (A) is used to encrypt and the other key (B) is used to decrypt. A message encrypted with *key A* cannot be decrypted with the same key. It must be decrypted with the *key B*. It is based on *Euler's theorem* that comes from number theory.

According to it, for each relatively prime numbers M and n , where $M < n$, the $M^{\Phi(n)} = 1 \pmod{n}$ is true.

How it works:[2]

If we have a sender (called "A") and a receiver (called "B").

1. First two prime numbers p and q must be chosen by "B" and then multiply them together to give n .
2. Then, "B" chooses a random natural number e which satisfies Euler's theorem with $\phi(n)$. The numbers n and e are the public key which "B" publishes.
3. To decrypt the message, "B" solves the equation $d = e^{-1} \pmod{\phi(n)}$ where d is the private key.
4. Now "A" wants to send a message to "B". For that purpose, "A" uses "B"'s public key e to cipher the message. The message is ciphered by solving the equation $c = m^e \pmod{n}$. c is the ciphertext that it's sent to "B"
5. Then the receiver "B" can decrypt the message by solving $m = c^d \pmod{n}$. Where m is the original text which "A" wanted to send.

3.6.2 Algorithm's security

Since n could be any size, RSA is a variable-length-key algorithm, therefore, the longer key, the more secure it is. Since the decryption does not take so much time, a larger size than the one used in *DES* is needed to avoid *brute force* attacks. This technique consists on trying all possible combinations.

This technique can be used for keys with less than 256 bits long. To attack a key of 256 bits with brute force, the attacker must try an average of 2^{255} keys which takes 10^{76} in time[2], therefore the system is quite brute-force-safe with big keys. Nowadays the common key size is 1024 bits.

There is one public-key-only attack called *factorisation attack*[2] which involves an attempt to invert a one-way function. RSA relies on the derivation of p and q from n which is a one-way function. The attack depends on the method used to find the factors of n . However, none of the previously known methods of factorisation are powerful enough to factorise a 1024-bit number with a reasonable cost.

Many RSA implementations use small values for e to enable a faster encryption. There is another attack called *low-exponent attack*[2]. When the e is small and there are an e number of receivers, all of them with messages encrypted with the same e ,

then, the *low-exponent attack* can regenerate the ciphertext. Even if only parts of the message are identical, it is still possible in part.

Therefore, RSA is breakable if it is poorly implemented. Too short key length, too small e parameter and other mistakes can lead to security holes.

If those mistakes are avoided, the RSA algorithm is a very good cipher algorithm.

3.6.3 Practice analysis

In our practical work, we have measured the time consumption during the encryption and decryption. Since the OS eventually expropriates the CPU, we have done rounds of 1.000 and 10.000 repetitions and then, we show the average of them. In this particular study we had to do less repetitions because this algorithm spends much more time during the encryption/decryption.

The measured times are the following :

Encryption :			
Message Size	Encrypted Message Size	1.000 repetitions	10.000 repetitions
56 bits	1024	101823229 ns	101801361.4 ns
64 bits	1024	102683435 ns	102062022.1 ns
128 bits	1024	102233199 ns	101824917.5 ns
256 bits	1024	102282821 ns	102501515.9 ns
512 bits	1024	101288267 ns	101694501.8 ns
1024 bits	2048	207847916 ns	202966183.7 ns
Decryption :			
Message Size	1.000 repetitions	10.000 repetitions	
56 bits	2786222 ns	2782930.8 ns	
64 bits	2780370 ns	2783808.6 ns	
128 bits	2783279 ns	2782788.5 ns	
256 bits	2778551 ns	2783236.6 ns	
512 bits	2781796 ns	2783501.7 ns	
1024 bits	5565539 ns	5561698 ns	

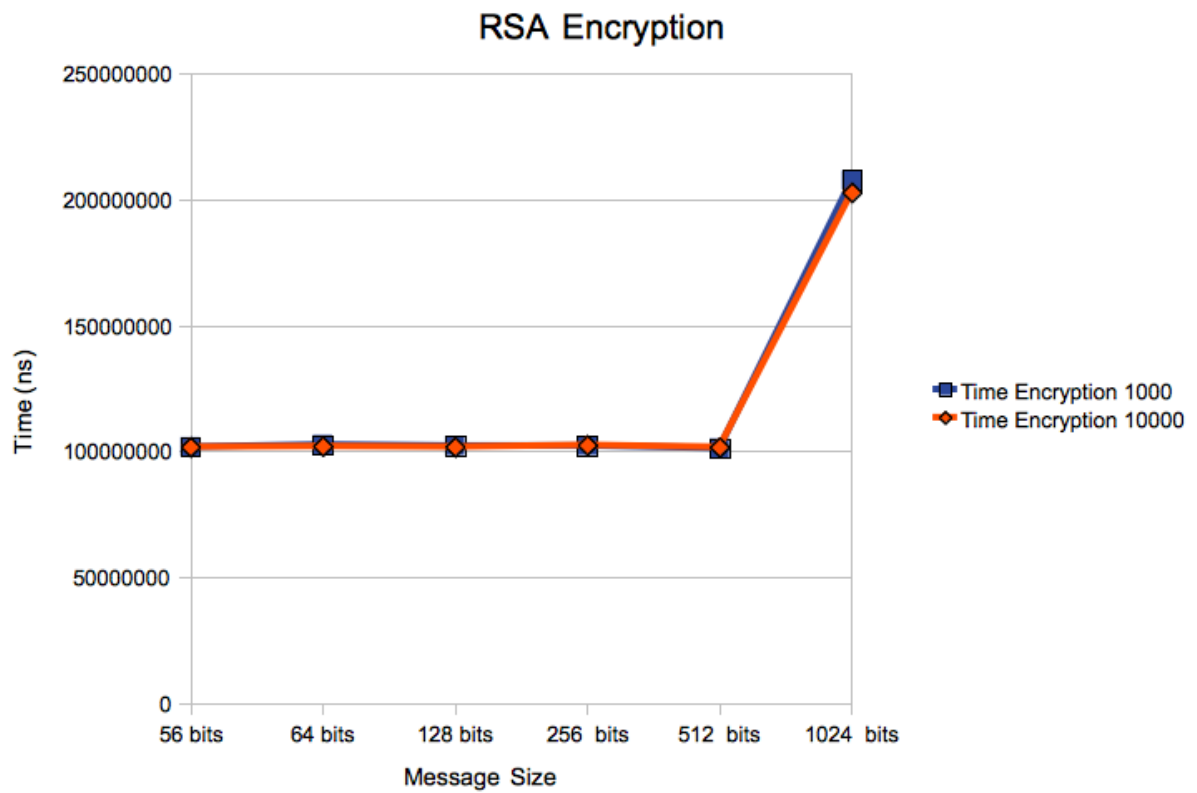


Figure 3.6.1: RSA Encryption

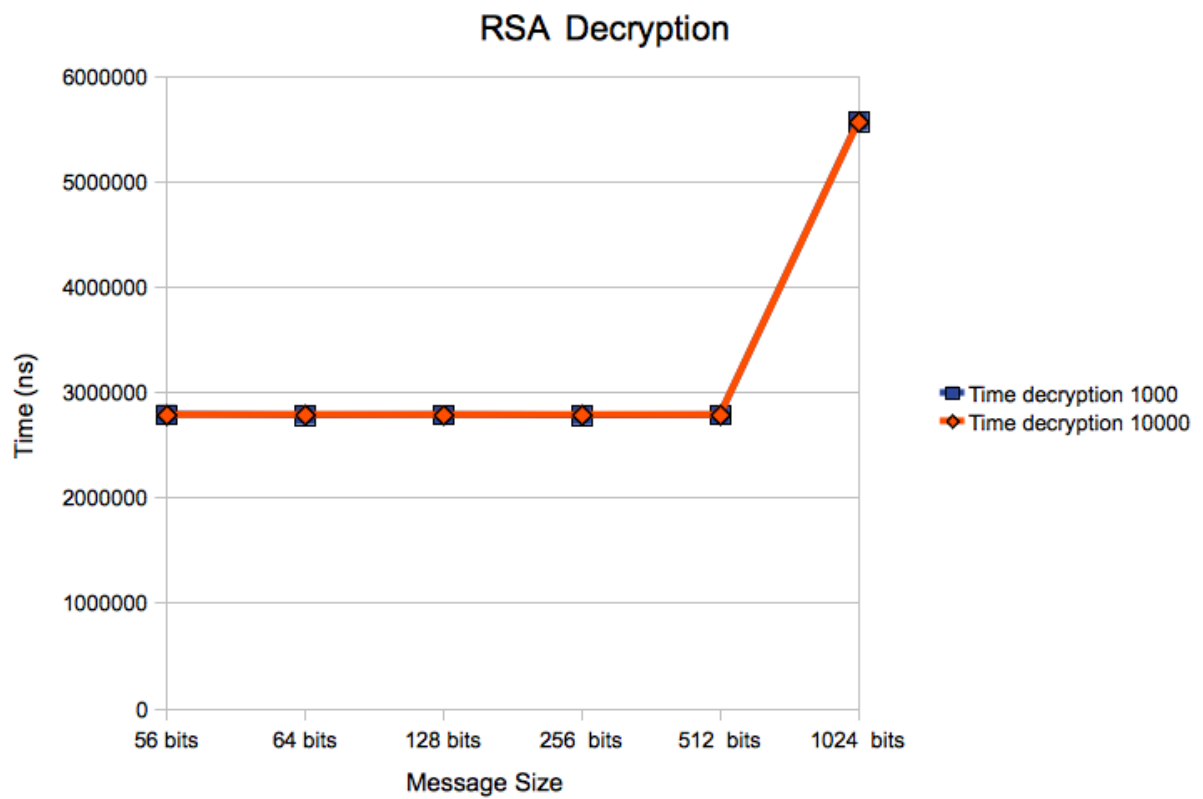


Figure 3.6.2: RSA Decryption

3.7 Diffie - Hellman key exchange

Diffie - Hellman key exchange is a cryptographic protocol that allows to establish a shared secret key between the sender and the receiver. Published by *Whitfield Diffie* and *Martin Hellman* in 1976.

The shared secret key is establish without a prior knowledge between the sender and the receiver. This key can be used to encrypt subsequent communications using a symmetric key cipher, such as the ones described in this paper.

3.7.1 Theoretical description

The original implementation of the protocol uses the multiplicative group of integers modulo p .

A prime number (p) and a primitive root(g) mod p are needed.

Let S be the sender and R the receiver, the key establishment is as follows:[14]

1. S and R agrees to use a prime number p and a base g
2. S chooses a secret integer a and sends to R : $A = g^a \mod p$
3. R chooses a secret integer b and sends to S : $B = g^b \mod p$
4. S computes $s = B^a \mod p$
5. R computes $s = A^b \mod p$

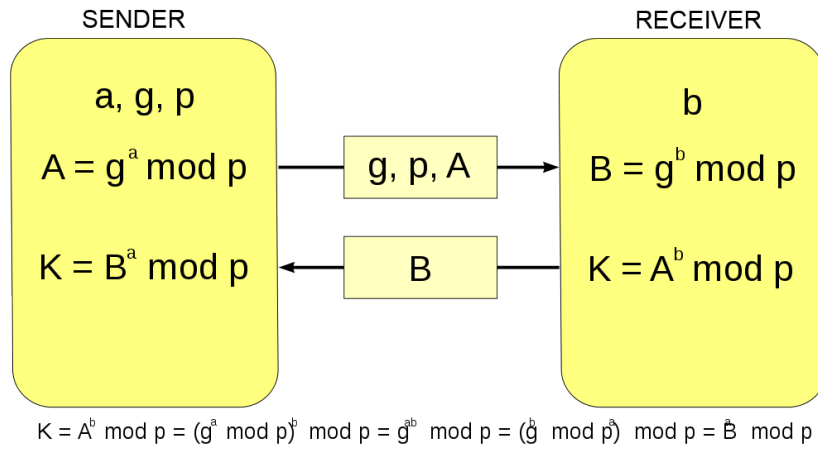


Figure 3.7.1: Diffie-Hellman key exchange[40]

Both have reached to the same value, because g^{ab} and g^{ba} are equal mod p . During the key setting, none of the messages have been encrypted, but since they have their own secret values, the key is perfectly safe and they are the only ones to know it.

To increase the security, large values of a , b and p may be used, because is easy to try all the values. For example if $p=13$ there will be, at most, 12 such values, no matter how big are a and b . The best way is to use random number generators for them, but if a not completely random number generator is used, they can be predicted to some extent, therefore, making easier to obtain the shared secret key.

Concerning security, the protocol is considered secure if G and g are chosen properly. A listener would have to solve the *Diffie-Hellman problem* to obtain g^{ab} , which is currently considered difficult.

An efficient algorithm to solve the discrete logarithm problem would make it easy to compute a or b and solve the *Diffie-Hellman problem*, therefore making this and many other public key systems insecure.

They can be also insecure, to have an "*man-in-the-middle attack*", if the listener establish two different Diffie-Hellman key exchanges, one with the sender and the other with the receiver. The attacker would be able to decrypt then re-encrypt the messages passed between the sender and the receiver.

In general, a method to authenticate the communicating parties to each other is generally needed to prevent it.

3.7.2 Algorithm's security

Diffie-Hellman protocol is generally considered safe when used appropriate mathematical groups. Maurer[33] has shown that breaking the Diffie-Hellman protocol is equivalent to computing discrete logarithms under certain assumptions.

Some discrete algorithms can be used to do attacks at Diffie-Hellman. Nowadays the passive attacks is the best way to attack Diffie Hellman at the moment on our analysis, assuming the correct election of the parameters.

The "man in the middle attack" also has effective results on the first version of this protocol. In this attack, another person can intercept "A" public value and send his own public value to "B". When "B" sends his public value, the attacker substitutes it with his own and sends it to "A". Then the attacker and "A" agree on one shared key and the attacker and "B" agree on another shared key.

After this exchange, the attacker simply decrypts any messages and then reads and possibly modifies them before re-encrypting with the corresponding key and transmitting to the corresponding receiver. This can be possible because Diffie-Hellman key exchange does not authenticate the participants.

The authenticated Diffie-Hellman key agreement protocol (*Station-to-Station protocol*) was developed to avoid the *man-in-the-middle attack*[34]. The immunity is achieved by allowing the two parties to authenticate themselves to each other by the use of digital signatures and public-key certificates.

3.7.3 Practice analysis

This practical analysis is different than the others. The times shown here are called encryptions but they have all the time amount of creating the key exchange. Therefore we only see one table.

Since this algorithm is too slow, we had to reduce the repetitions down to 1.000. The computer needs more than 30 minutes to do this amount of repetitions, therefore any possible interruption of the Operative System will be already included in the average. Since we are going to compare this algorithm with the RSA algorithm, both algorithms will be under the same situation and the same repetitions amount. In this way we do not bias one algorithm or the other.

Encryption :	
Message Size	1.000 repetitions
56 bits	243363288 ns
64 bits	234757718 ns
128 bits	246706889 ns
256 bits	241498153 ns
512 bits	234170736 ns
1024 bits	237722762 ns

4 Discussion

Once we have introduced all the algorithms, the advantages and disadvantages of each one are :

1. DES :

In 1976 was selected by the *National Bureau of Standards* as an official *Federal Information Processing Standard (FIPS)* for the United States. This algorithm is broken, therefore is an insecure algorithm and it cannot be in our application.

2. 3DES :

3DES has a really high time consumption and with his short key problems described in 3.2.2 is not a good candidate for our application.

3. Serpent :

This algorithm is safe and efficient. Under some implementations it's faster than AES that's to the code optimization. In our study, since we are dealing with a Java application, we have used the Java version and it's slower than AES.

4. AES :

This algorithm was introduced as a standard in 2001 after 5 years of standardization after being in a contest with more algorithms. Thanks to those 5 years, it has been deeply studied and adopted as a standard by the U.S. Government. As we described in section 3.3.2 there is no possible attack to break the algorithm yet. In our practical test we have measured the time consumption of encryption and decryption and only the Blowfish algorithm has beaten those times (by a really slight difference)(Figure 4.0.3).

5. Blowfish :

There's no vulnerability found yet and this algorithm has the best time consumptions of all the symmetric algorithms in our study. Only a brute force attack can beat it but it takes a huge amount of time and data.

6. RSA :

This is the most know asymmetric algorithm. RSA efficiency cannot be compared to the symmetric algorithms, but this algorithm is really useful if you want to be *man-in-the-middle attack* safe. For this purpose we decided to use it in our client-server application to establish a symmetric key between the client and the server.

7. Diffie Hellman :

This key exchange protocol is not interesting for us in his initial version, since it is *man-in-the-middle attack* unsafe. Even if it is a safe algorithm, the time consumption of Diffie Hellman is worse than the RSA, therefore is not from our interest to achieve a better performance.

The following graphs show the same test ran on each algorithm, to show us how different or not they are:

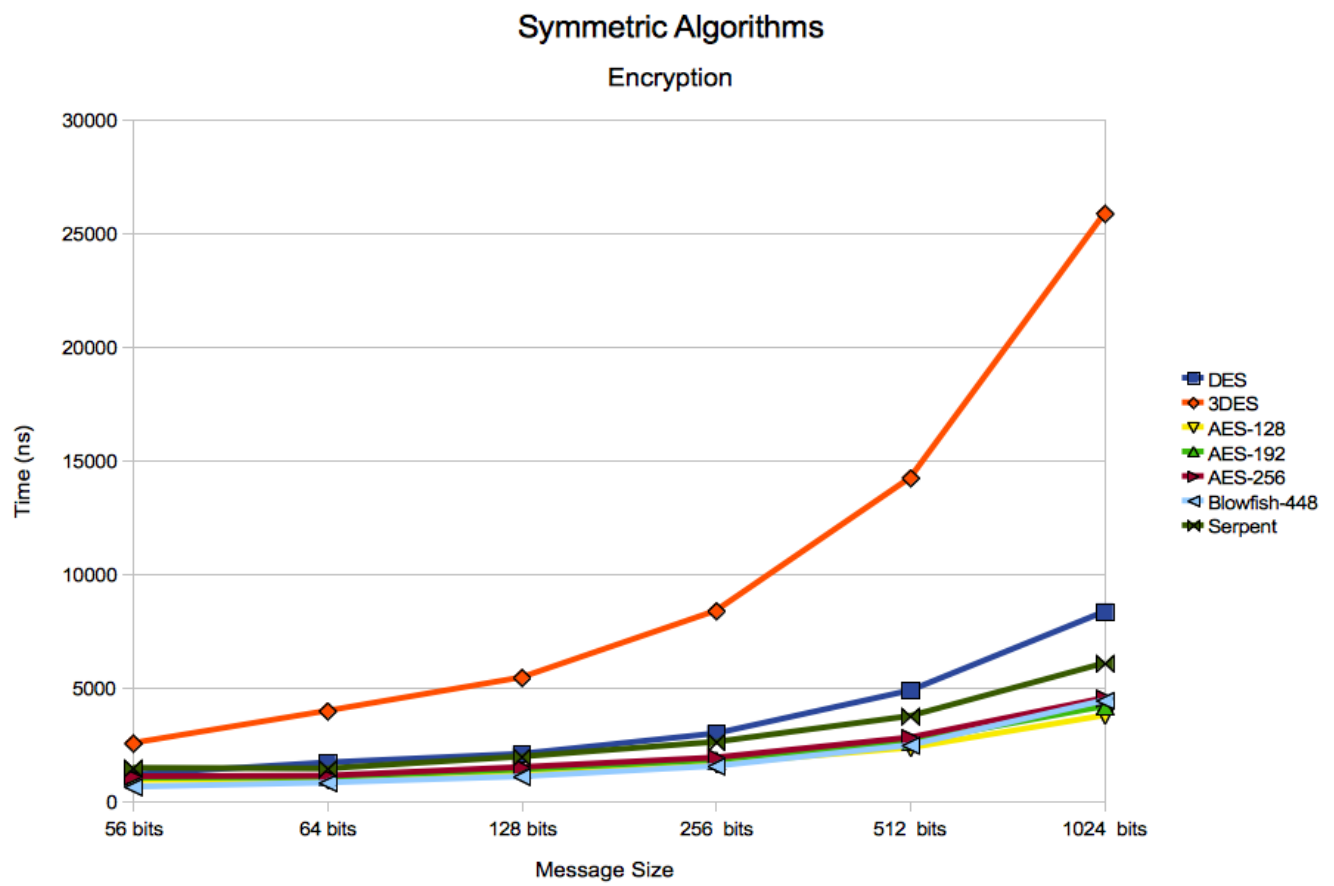


Figure 4.0.1: Encryption of Symmetric Algorithms

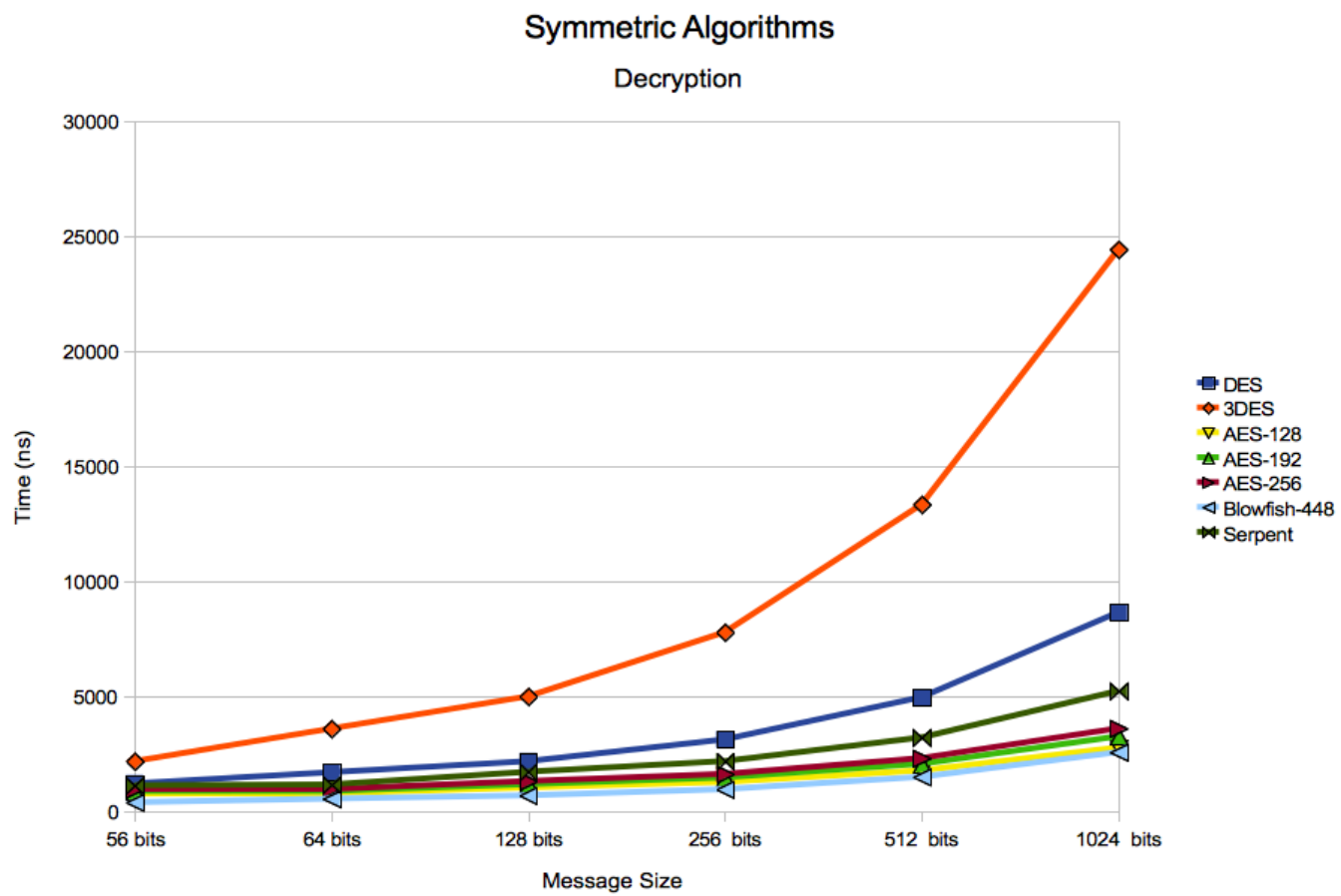


Figure 4.0.2: Decryption of Symmetric Algorithms

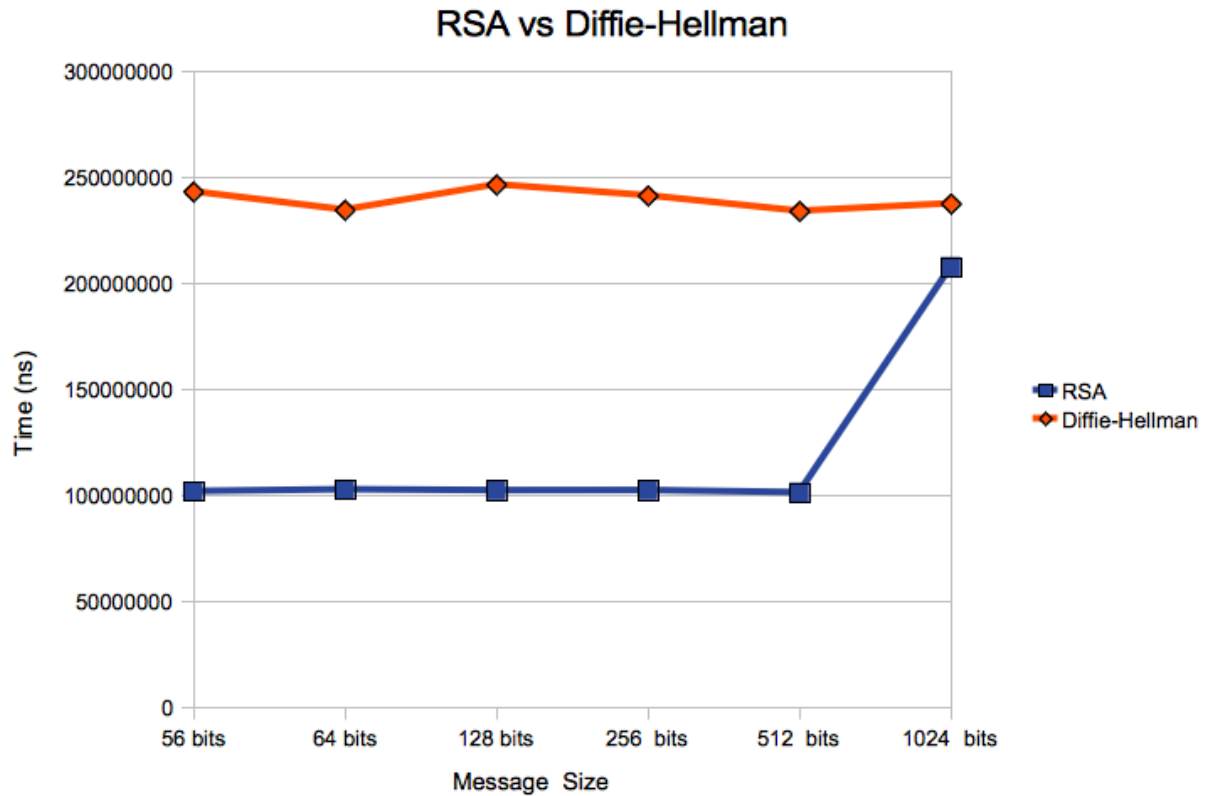


Figure 4.0.3: Asymmetric Algorithms

Final choice :

To ensure a secure communication we need to cypher all the data that we want to send though the insecure channel (the Internet) with some encryption method. The symmetric encryption is much faster than the asymmetric encryption, nevertheless we need to establish a symmetric key between the client and the server to start the communication. For that purpose, we are going to use RSA encryption for the key exchange and then we will use the Blowfish encryption method to communicate between the client and the server. This will be secure enough because on each connection between the client and the server, we are using a different encryption key and the sessions will not be long enough to perform a bruce force attack.

The communication process will be :

1. The client generates a Blowfish key for the communication.
2. The client cyphers the Blowfish key with the public RSA key from the server.
3. The server receives the cyphered data, decrypts the data with his private RSA key and obtains the Blowfish Key.
4. The server will communicate the reception to the client and the communication process from here will be encrypted with the Blowfish key.

In this way we are sure that our communication will be secure.

The reasons of choosing Blowfish as our main encryption algorithm are, first, the efficiency on time consumption, and second, there is no effective attack know

against it. Therefore is a fast and safe algorithm. Before choosing Blowfish we considered the AES algorithm. AES has a time consumption close to Blowfish and maybe the AES security is more proved since it is more used and it is a standard, but we choose Blowfish because the symmetric keys are only going to be used during one communication session, therefore the Blowfish security is enough for us and it is faster.

5 Java study application

We have built a Java application to study and test all the algorithms. The client-server application is Java based and the encryption algorithms are going to be written in Java, therefore, our test application is Java based too.

Libraries used from Bouncy Castle[42]:

- bcpmail-jdk16-145.jar
- bcpg-jdk16-145.jar
- bcprov-jdk16-145.jar
- bctsp-jdk16-145.jar
- jce-ext-jdk13-145.jar
- jce-jdk13-145.jar

Bouncy Castle is a collection of APIs used in cryptography. It includes APIs for both the Java and the C# programming languages. It is a free software.

We also needed to update two security libraries[43] the main Java path in order to use some of the algorithms:

They are located in Java/Home/lib/security/

1. US_export_policy.jar
2. local_policy.jar

5.1 Class Diagram

In Figure 5.1.1 the general schema is shown. The schema follows the Model–View–Controller (MVC) architecture. The Model is the specific representation of the information with the system works. The View represents the user interface where the information is represented in a way that the user is able to understand. The Controller reacts to the events and it invokes petitions to the model.

The View has three different classes :

1. Vista : Is the main window of the application.
2. Study : Window where the number of repetitions is introduced.
3. Result : Window where the study results will be shown.



5.2 Package Hierachy

This application follows the Model–View–Controller (MVC) architecture.

5.2.1 Default Package

This package only contains the Java class with the main method to start the application.

5.2.2 Model Package

- **Modelo Class** : This class has the application data. This class contains an observer (implemented by Vista class) to inform about any changes.

5.2.3 Controlador Package

- **Controlador Class** : This class gives to the model the neccesary intructions to have a proper behaviour, therefore, the application works as it should.

5.2.4 Vista Package

- **Vista Class** : This class implements the Observador interface. It also builds the main window and this class informs about all the actions taken by the user.
- **Result Class** : This class is in charge of representing all the results of the chosen study.
- **Study Class** : This class launches a study with the introduced data in the main window and the number of repetitions.

5.2.5 Observador Package

- **Observador Interface** : Interface which declares all the methods used to inform the model about any changes.

5.3 User Manual

Here we describe how to use the study application.

5.3.1 Main Window

On the main window (Figure 5.3.2) we can see all the available algorithms for the study.

The *key* field is to write a desired key to introduce on the algorithm.

The box under the *key* field is to write down the text to encrypt/decrypt.

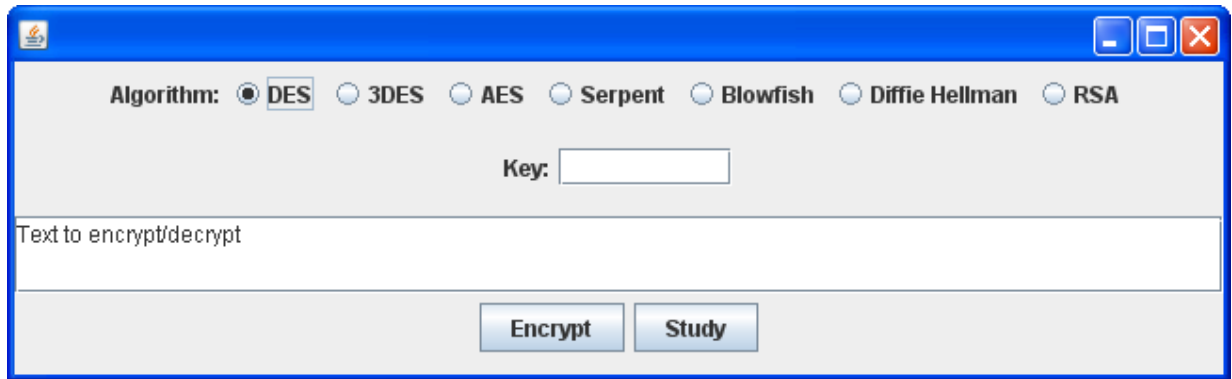


Figure 5.3.2: Study Application - Main Window

Then we can find 2 buttons on the bottom of the window.

- **Encrypt** : This button will only execute one encryption of the introduced text with the desired key. When the application has encrypted and decrypted the data, another window will be displayed (Figure 5.3.3)
- **Study** : Once the key and the text to encrypt is written, the *Study* button opens a new window where you can type how many repetitions you want to execute.

5.3.2 Result Window

This window (Figure 5.3.3) show the result of one or more encryptions.

The original text is shown in bits, hexadecimal and in chars on the 3 text boxes of the top.

Under them, there are another 3 boxes to show the encrypted text in bits, hexadecimal and in chars too.

Under the boxes, on the left, we can see the total amount of time of all encryptions and decryptions. There is also the encrypted text size.

On the top of the window, the name of the algorithm used is written.

To go back to the main window we only have to press the *Close* button.

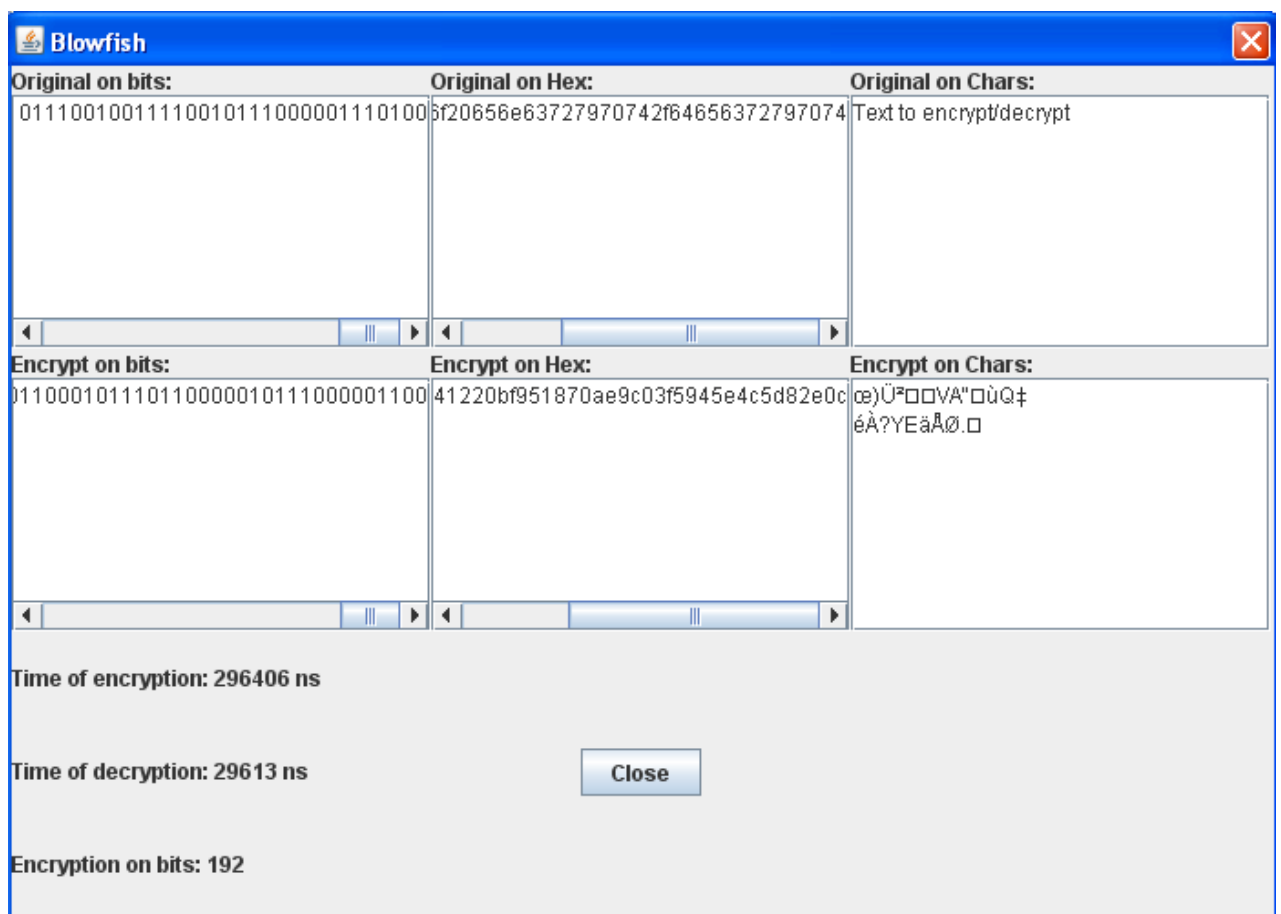


Figure 5.3.3: Study Application - Result Window

6 Client-Server application

There are two different applications inside, the Server and the Client. Everything has been designed with the Model–View–Controller (MVC) architecture.

Once we have finished the algorithm study, we are going to use all the acquired knowledge in a real application. For that purpose we want to create a client-server application.

The clients will connect to a server and they will share some private information. Therefore we need a confident communication process. The client has to log in with his username and password.

For this purpose we are going to use two different encryption types : symmetric and asymmetric encryptions. First of all, the client sends a public key and the server will reply with a symmetric key cyphered with the public key. Then the client will decrypt the received message to know the symmetric key to send all the information to the server.

Now both know the key to encrypt all the information during the communication.

We want to build an application very close to the real world, so it has to work over a network (such as the Internet) and with a database server. The database server provides all the necessary data for the application.

Our application is a shopping application. The servers are restaurants that provides some types of food and the clients will choose from which restaurant and which food they want to buy. All the restaurants will act as administrators, choosing which communication port they want to use to wait for the clients. The clients have to choose the ip-address and the port for the server to make the connection.

The application must be able to manage all the database information, password checking and the business intelligence such as reports, database checks, etc.

6.1 Class diagrams

The class diagrams of this application are :

6.1.1 Client Diagrams

In Figure 6.1.1 the general schema is shown. The schema follows the Model–View–Controller (MVC) architecture. The Model is the specific representation of the information with the system works. The View represents the user interface where the information is represented in a way that the user is able to understand. The Controller reacts to the events and it invokes petitions to the model.

In Figure 6.1.2 the main Model structure is shown. The main class of the Model is "Modelo" which has a thread (Hilo) which is going to send to the server and receive them from it. The encryption and decryption is also managed there. Both Modelo and Hilo use the different classes which represents the messages that can be send/received for our application.

Each message can be :

1. AgregarTextoSala : Sends a text message to a specific chatroom.
2. ConfirmacionPedido : Confirms an order.
3. EntrarSala : Informs about a user which is entering a chatroom.
4. ExisteUsuario : Informs if a specific user is in our chat.
5. FinPrivado : Informs about the end of a private chat.
6. ListaSalas : Informs about the available public chatroom list.
7. ListaUsuariosConectados : Informs about the existing users in a specific chatroom.
8. MensajePrivado : Sends a private message to a private chatroom.
9. Pedidos : Sends the information of an user order.
10. PeticionConexion : Requests the connection to a server.
11. PeticionDesconexion : Informs about the disconnection from the server.
12. PlatosCategorias : Requests the dish information sorted by categories.
13. PlatosRestaurantes : Requests the dish information group by restaurants.
14. SalasUsuario : Requests the chatroom list where a specific user is in.
15. SalirSala : Informs about an user exiting a chatroom.
16. UsuariosSala : Requests the user list of a chatroom.

VistaUser has a class called *Carrito* which represents the selected products to order and it will use the class *PanelTabla* to represent the product lists. There is also another class called *VentanaPrincipal* which represents the chat on the application. *VentanaPrincipal* has some *PanelChat* instances to represent the public chat conversations and the *PanelChatPrivado* will represent the private ones.



In Figure 6.1.4 the general schema is shown. The schema follows the Model–View–Controller (MVC) architecture. The Model is the specific representation of the information with the system works. The View represents the user interface where the information is represented in a way that the user is able to understand. The Controller reacts to the events and it invokes petitions to the model.



The *Usuario* class has a class called *HiloCliente* to send and receive the messages of an specific client. The encryption and decryption will be managed by this class.



The Figure 6.1.6 shows a diagram more focused on the message passing treatment between the client and the server.

The class *HiloCliente* and *Modelo* uses a series of classes which represent the possible messages:

1. *AgregarTextoSala* : Sends a text message to a specific chatroom.
2. *ConfirmacionPedido* : Confirms an order.
3. *EntrarSala* : Informs about a user which is entering a chatroom.
4. *ExisteUsuario* : Informs if a specific user is in our chat.
5. *FinPrivado* : Informs about the end of a private chat.
6. *ListaSalas* : Informs about the available public chatroom list.
7. *ListaUsuariosConectados* : Informs about the existing users in a specific chatroom.
8. *MensajePrivado* : Sends a private message to a private chatroom.
9. *Pedidos* : Sends the information of an user order.
10. *PeticiónConexion* : Requests the connection of a new client to the server.
11. *PeticionDesconexion* : Informs about the disconnection of a client.
12. *PlatosCategorias* : Requests the dish information sorted by categories.
13. *PlatosRestaurantes* : Requests the dish information group by restaurants.
14. *SalasUsuario* : Requests the chatroom list where a specific user is in.
15. *SalirSala* : Informs about an user exiting a chatroom.
16. *UsuariosSala* : Requests the user list of a chatroom.

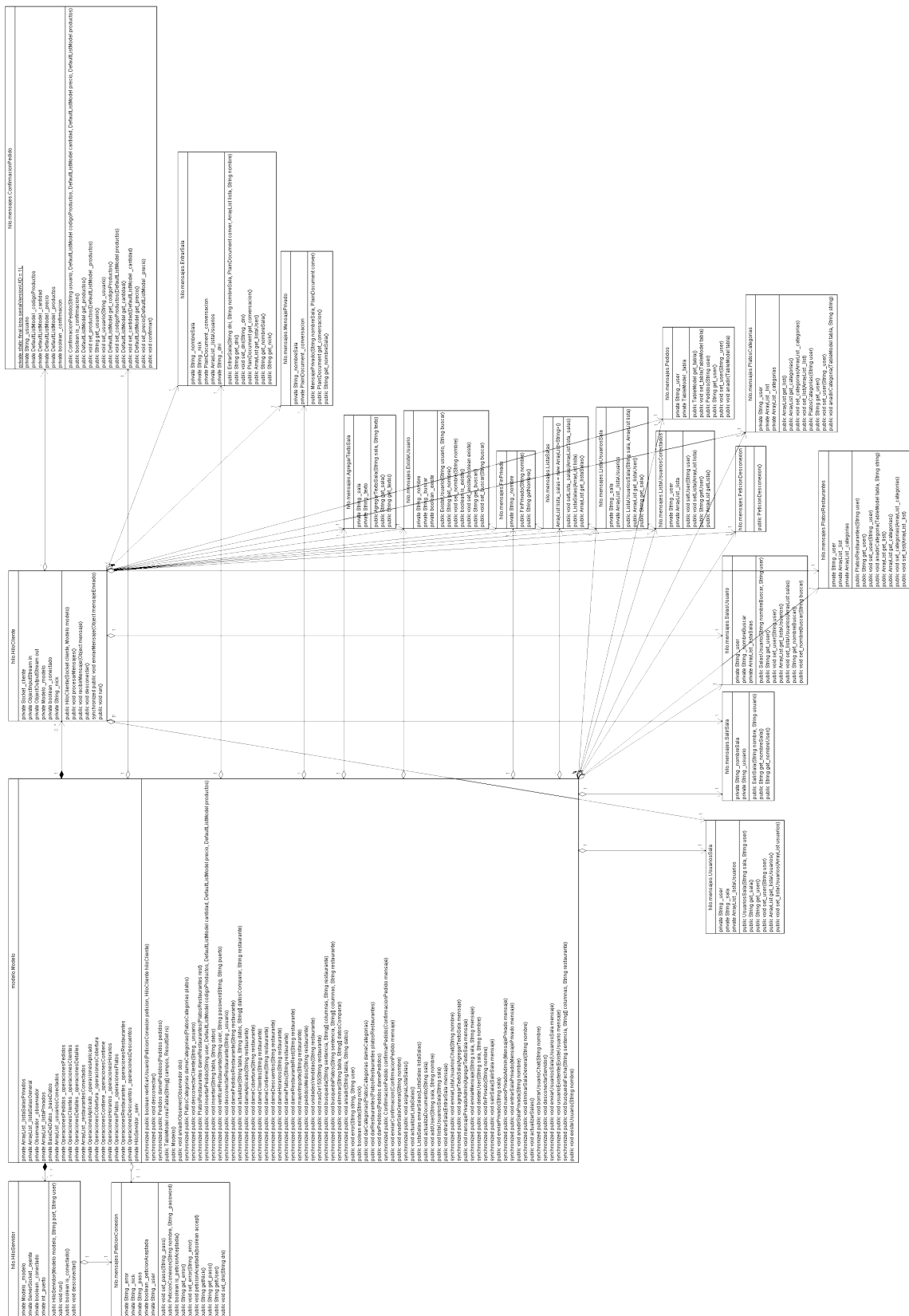


Figure 6.1.6: Server Application - Hilos Diagram

The *VistaRestaurante* class has the classes *VentanaPrincipal* (the main chat window), *VentanaInforme* (shows the desired report), *PanelBusquedaPlatos* (to search for dishes), *PanelBusqueda* (to search a person), *PanelBusquedaFecha* (to search orders from a specific month) and *FormularioModificar* (to create a modification form for a specific table).



In Figure 6.1.8 the specific diagram part for the View to represent the chat is shown. The main class is *VentanaPrincipal* and has the *NuevaSala* class (to create a new public chatroom), *PanelChat* (to represent the public chatroom), *PanelChatPrivado* (to represent the private chatroom). There is also a class named *BotonPestana* to be able to close the chat conversations.

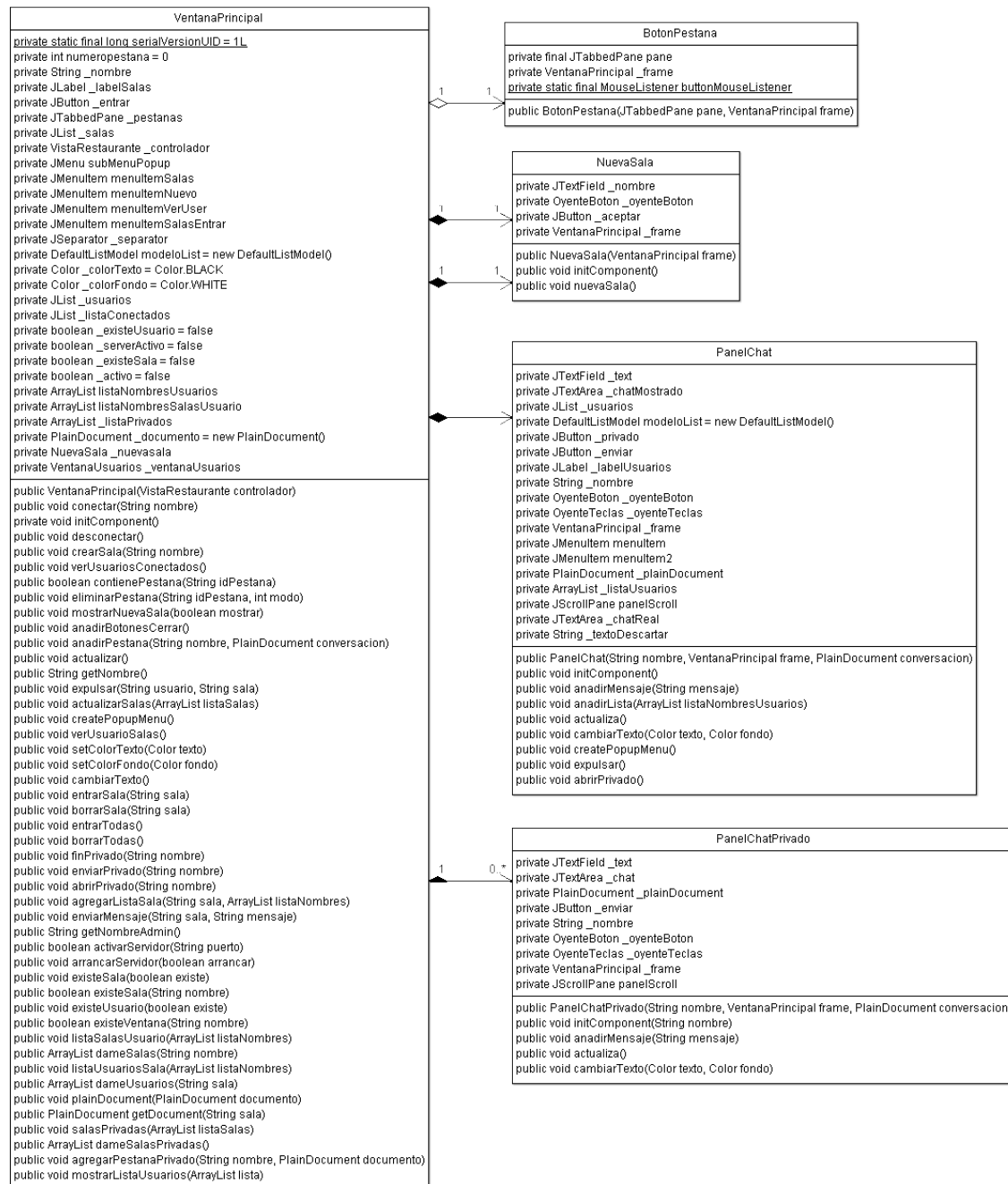


Figure 6.1.8: Server Application - Vista Diagram with Chat

6.2 Server

The *Aplicacion Package* (Application Package) contains the starting point for the server application.

The *Modelo Package* (Model Package) contains the classes to store the data.

The *Controlador Package* (Controller Package) contains the logic to update and interpret the model's data. In this case it's more like an intermediate between the view and the model.

The *Vista Package* (View Package) contains the necessary classes to build a window-based user interface and it has the necessary classes to capture the data introduced by the user.

The *Hilo Package* (Thread Package) is in charge of the communications between the Server application and the Client application. It also does the data encryption and decryption.

The *Observador Package* notifies the changes to the view.

There are other sub-packages :

1. `modelo.operaciones` : with classes to do specific operations with the tables
2. `vista.restaurante` : User's restaurant view
3. `vista.formularios` : Panles where the data from the database is shown
4. `vista.chat` : Chat's interface in the application.
5. `hilo.mensajes` : Messages sent throught the thread to the clients

6.2.1 Aplicacion Package

This package has only one Java Class with the Main to start the application. The model is generated, then the controller which has a model assigned to work with and at the end the view is created. After that, an observer called *vista* is introduced in the model, to capture the received changes.

```
public class Aplicacion {  
    //The model is created.  
    Modelo _modelo=new Modelo();  
    //The controller is created.  
    Controlador _controlador=new Controlador(_modelo);  
    //The view is created.  
    Vista _vista=new Vista(_controlador);  
    _modelo.anadirObserver(_vista);  
}
```

6.2.2 Modelo Package

1. **Modelo Class** : This class has the representation of the business information. There's a `Modelo` class (model) that contains all the necessary data for the application. This class has an obverser (`Vista`) which is going to be noticed about any changes on the model.
2. **Basededatos Class** : This class is going to manage our database.

3. Tablemodel Class : This class is going to build the table where the desired data is going to be shown.
4. Salageneral Class : Represents a general chat room.
5. Salaprivado Class : Represents a private chat room.
6. Usuario Class : Represents a chat user.
7. Modelo.Operaciones Package : Created to deal with specific operations.
 - (a) Operacionespedidos Class : Deals with the pedidos table
 - (b) Operacionesplatos Class : Deals with the platos table
 - (c) Operacionesrestaurantes Class : Deals with the restaurantes table
 - (d) Operacionesclientes Class : Deals with the clientes table
 - (e) Operacionesaplicado Class : Deals with the aplicadoa table
 - (f) Operacionescobertura Class : Deals with the cobertura table
 - (g) Operacioneshorarios Class : Deals with the horarios table
 - (h) Operacionesdescuentos Class : Deals with the descuentos table
 - (i) Operacionescontiene Class : Deals with the contiene table

6.2.3 Controlador Package

This package has only the Controlador Class to act as a bridge between the view and the model.

6.2.4 Observador Package

Here we can find the *observador* interface which is implemented by the vista class (the view in the MVC). This interface is set up to establish a number of methods which are going to be watching the application events and they will update the view when a change is done.

6.2.5 Vista Package

1. Vista Class : This class is going to control all the application windows. It implements the *observador* interface to be able to receive all the necessary data from the changes on the model. It also contains the login window for the restaurants.
2. Ventanaloginrest Class : Class to implement the restaurant login window.
3. Vista.Restaurante Package
 - (a) Panelbusqueda Class : Panel where the user is able to check all the information related to one specific client. It can be sorted by a specific category.
 - (b) Panelbusquedaplatos Class : It builds the panel where all the information associated to one product is displayed.

- (c) Panelbusquedafecha Class : It creates the panel with all the orders done in an specific month.
- (d) Ventanainforme Class : It creates a window where the report is shown and ready to print.
- (e) Vistarestaurante Class : This class is going to create and control the restaurant interface. It has a *JTabbedPane* where all the asked information will appear.

4. Vista.Chat Package

- (a) Botonpestaña Class : Class in charge of representing the button on each chat room to close and leave the chat room.
- (b) Nuevasala Class : It creates a window to type in the name of the new chat room.
- (c) Panelchat Class : This class represent the panel of a general chat room.
- (d) Panelchatprivado Class : This class represent the panel of a private chat room.
- (e) Ventanaprincipal Class : This class builds the panel which represents the chat.

6.2.6 Hilo Package

1. Hilocliente Class : Receives and sends the information to each client. The encryption and decryption is done in this class.
2. Hiloservidor Class : This class creates the execution thread which connects the server with an specific port to be listened. The petitions from the clients will arrive from the clients to that port. When there is a valid connection petition, a new HiloCliente will be launched for the data exchange between the client and the server, allowing multiple connections at the same time.

3. Hilo.mensajes Package

- (a) Agregartextosala Class : This class requests to add a message to one chat room.
- (b) Confirmacionpedido Class : It is used to confirm a client order.
- (c) Entrarsala Class : This class requests the login to a client into a specific chat room.
- (d) Existeusuario Class : This class checks if some username exists.
- (e) Finprivado Class : This class informs about the end of a chat room.
- (f) Listasalas Class : This class requests a chat room list.
- (g) Listausuariosconectados Class : This class request the connected user list.
- (h) Listausuariossala Class : This class sends the user list of an specific chat room.
- (i) Mensajeprivado Class : This class requests a private conversation.

- (j) Pedidos Class : This class request the orders of an specific client.
- (k) Peticiondeconexion Class : This class request the connection of a client.
- (l) Peticiondesconexion Class : This class request the disconnection of a client.
- (m) Platoscategorias Class : This class request a list of dishes sorted by categories.
- (n) Platosrestaurantes Class : This class request a list of ordered dishes by restaurants.
- (o) Salasusuario Class : This class request a list with all the chat rooms where an specific user is in.
- (p) Salirsala Class : This class requests to some client to leave the room.
- (q) Usuariossala Class : This class requests the user list of an specific chat room.

6.3 Client

The *Aplicacion Package* (Application Package) contains the starting point for the client application.

The *Modelo Package* (Model Package) contains the classes to store the information.

The *Controlador Package* (Controller Package) contains the logic to update and interpret the model's data. In this case it's more like an intermediate between the view and the model.

The *Vista Package* (View Package) contains the necessary classes to build a window-based user interface and it has the necessary classes to capture the data introduced by the user.

The *Hilo Package* (Thread Package) is in charge of the communications between the Server application and the Client application. It also does the data encryption and decryption.

The *Observador Package* notifies the changes to the view.

There are other sub-packages :

1. vista.chat : The Chat interface in the application.
2. vista.usuario : Represents the main window for the client application.
3. hilo.mensajes : Messages sent through the thread to the servers.

6.3.1 Modelo Package

1. Modelo Class : In this class we can find the application data. This class has an observer (Vista) which is going to be informed about changes in the model.
2. Tablemodel Class : This class represents the table in which the query results to our database are going to be shown.
3. Salageneral Class : Represents a general chat room.
4. Salaprivado Class : Represents a private chat room.

6.3.2 Controlador Package

1. Controlador Class : In this application is only a bridge between the view and the model.

6.3.3 Observador Package

1. Observador Interface : Interface implemented by *Vista*.

6.3.4 Vista Package

1. Vista Class : This class controls all the windows of the application. Implements the *Observador* interface to be able to receive the necessary information about the changes on the model. It also has the login windows for the clients and they will be shown only when is necessary.
2. Pedirdatos Class : Implementation class for the client access window.
3. Vista.Usuario Package :
 - (a) Carrito Class : This class creates a panel were we will be able to see the shopping cart of our application. We are able to see the cumulative cost, which products and how much quantity of each. A price per unit is also visible. There are 2 JButtons to confirm the order or to empty the shopping cart. If we use the right button on the screen, a secondary menu will be displayed to be able to delete or modify the selected product quantity in the shopping cart. There's also a clock on the bottom of the panel.
 - (b) Paneltabla Class : It shows a panel with a table (*JTable*) where the previously requested information is going to be showed.
 - (c) Reloj Class : This class creates a clock on the bottom of the window. It also shows the time when you logged in.
 - (d) Vistauser Class : This class creates and controls the client interface. It has a *JTabbedPane* where all the information will appear.
4. Vista.Chat Package :
 - (a) Botonpestaña Class : It builds a button on each tab of our chat. With it you are able to close the tab and exit the chat room.
 - (b) Nuevasala Class : This class shows a new window to introduce the name of the new chat room.
 - (c) Panelchat Class : This class represents the panel of a general chat room.
 - (d) Panelchatprivado Class : This class represents the panel of a private chat room.
 - (e) Ventanaprincipal Class : This class builds the panel to represent the entire chat.

6.3.5 Hilo Package

1. Hilo Class : This class is going to receive and send the data between the client and the server on an specific port. It also encrypts and decrypts the data.
2. Hilo.Mensajes Package :
 - (a) Agregartextosala Class : Requests the addition of a message to a chat room.
 - (b) Confirmacionpedido Class : Confirms the client order.
 - (c) Entrarsala Class : Request the entrance on an specified chat room.
 - (d) Existeusuario Class : This class verifies if an specific user exists.
 - (e) Finprivado Class : This class notifies the end of the private chat room.
 - (f) Listasalas Class : Requests a chat room list.
 - (g) Listausuariosconectados Class : This class sends the user list of an specific chat room.
 - (h) listausuariossala Class : Request a list of the users connected to an specific chat room.
 - (i) mensajeprivado Class : Request the start of a private conversation.
 - (j) pedidos Class : Requests the orders of a specific user.
 - (k) peticiondeconexion Class : Requests a login.
 - (l) peticiondesconexion Class : Requests a logout.
 - (m) platoscategorias Class : Requests a list of the dishes sorted by categories.
 - (n) platosrestaurantes Class : Requests a list of the dishes sorted by restaurants.
 - (o) salausuario Class : Requests the list of chat rooms where the user is in.
 - (p) salirsala Class : This class requests to some client to leave the room.
 - (q) usuariossala Class : This class requests the user list of an specific chat room.

6.4 Resources used

Libraries of Bouncy Castle[42]:

- bcprov-jdk16-145.jar

Data base: Practica4_LPS.mdb

To install the library on windows:

Control Panel ->administrative tools->data sources(ODBC)->add...->Driver do Microso

6.5 Encryption implementation on the applications

6.5.1 Client application :

When a Hilo class is created, the file containing the public RSA key will be opened and stored in a variable for further use.

```
try {  
    f = new FileInputStream("public.out");  
} catch (FileNotFoundException e1) {  
    e1.printStackTrace();  
}
```

```
ObjectInputStream o = null;
```

```
try {  
    o = new ObjectInputStream(f);  
} catch (IOException e1) {  
    e1.printStackTrace();  
}
```

```
try {  
    _publicKey=(PublicKey)o.readObject();  
} catch (IOException e) {  
    e.printStackTrace();  
} catch (ClassNotFoundException e) {  
    e.printStackTrace();  
}
```

```
try {  
    o.close();  
} catch (IOException e) {  
    e.printStackTrace();  
}
```

```
try {  
    f.close();  
} catch (IOException e) {  
    e.printStackTrace();  
}
```

Client connection step by step :

1. A Blowfish key is generated and encrypted with the RSA public key. Then the client sends the package to the server and waits for the reply.

```
public void run(){
try {
    int puerto=Integer.parseInt(_puerto);
    _socketCliente=new Socket(_dirIP, puerto);
    out = new ObjectOutputStream(_socketCliente.getOutputStream());

    KeyPairGenerator keyGen;
    SealedObject aux = null;
    try {
        Cipher cifrar = Cipher.getInstance("RSA/ECB/PKCS1Padding");

        cifrar.init(Cipher.ENCRYPT_MODE,_publicKey);

        KeyGenerator skf = KeyGenerator.getInstance("Blowfish");
        skf.init(128);
        _key=skf.generateKey();

        SealedObject so=
            new SealedObject(new String(_key.getEncoded()),cifrar);

        out.writeObject(so);

        in = new ObjectInputStream(_socketCliente.getInputStream());

        aux = (SealedObject)in.readObject();

    } catch (NoSuchAlgorithmException e) {
        e.printStackTrace();
    } catch (NoSuchPaddingException e) {
        e.printStackTrace();
    } catch (InvalidKeyException e) {
        e.printStackTrace();
    } catch (IllegalBlockSizeException e) {
        e.printStackTrace();
    }
}
```

2. When the server replies that he received the symmetric key, we send a connection request cyphered with the symmetric key.

```
PeticionConexion peticionRecibida=null;
```

```

        if(blowfishDec(aux) instanceof ClaveRecivida){
            PeticionConexion petition = new PeticionConexion(_nick,_password);
            SealedObject so=blowfishCyp(peticion);

            out.writeObject(so);

```

3. The client waits a server reply, if the reply is favorable, the client connects to the server and all the data sent and received will be processed with the symmetric key. If the connection is denied an error message will appear and the Hilo will close.

```

        petitionRecibida=null;
        petitionRecibida=
            (PeticionConexion) blowfishDec((SealedObject)in.readObject());
    }
    if ((petitionRecibida!=null)&&(petitionRecibida.is_peticionAceptada())){
        _conectado=true;
        _modelo.usuarioConectado(petitionRecibida.getUser());
        _peticionGestionada=true;
        procesarMensajes();
    }
    else{
        if(petitionRecibida==null)
            _modelo.mostrarMensajeError("Server is not operative. ","Error");
        else
            _modelo.mostrarMensajeError(petitionRecibida.get_error(),"Error");

        _conectado=false;
        _peticionGestionada=true;
        _socketCliente.close();
    }
    _servidorConectado=true;
} catch (ConnectException e) {
    e.printStackTrace();
    _modelo.mostrarMensajeError("Error of connexion with the server. ","Error");
    System.exit(0);
} catch (UnknownHostException e) {
    e.printStackTrace();
    _modelo.mostrarMensajeError("Error of connexion with the server. ","Error");
    System.exit(0);
} catch (IOException e) {
    e.printStackTrace();
    _modelo.mostrarMensajeError("Error of connexion with the server. ","Error");

    _peticionGestionada=true;
    _conectado=false;
    _servidorConectado=false;
    desconectar(false);

```

```

        System.exit(0);

    } catch (ClassNotFoundException e) {
        _modelo.mostrarMensajeError("Error de conexion con el servidor","Error");
        desconectar(false);
        System.exit(0);
    }
}

```

6.5.2 Server application :

When a HiloCliente class is created, the file where the RSA private key is stored (private.out) is opened, then it will be stored in a variable on the application.

```

FileInputStream f = null;
try {
    f = new FileInputStream("private.out");
} catch (FileNotFoundException e1) {
    e1.printStackTrace();
}

ObjectInputStream o = null;
try {
    o = new ObjectInputStream(f);
} catch (IOException e1) {
    e1.printStackTrace();
}

try {
    _private=(PrivateKey)o.readObject();
} catch (IOException e) {
    e.printStackTrace();
} catch (ClassNotFoundException e) {
    e.printStackTrace();
}

try {
    o.close();
} catch (IOException e) {
    e.printStackTrace();
}

try {
    f.close();
} catch (IOException e) {
    e.printStackTrace();
}

```


1. If a SealedObject arrives (object which contains the cyphered text), then is decrypted with our private key, then we have the symmetric key sent by the client.

```
public void run(){

    try {
        Object mensajePetición=in.readObject();
        if(( mensajePetición instanceof SealedObject )&&(!_conexión)){
            Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");

            cipher.init(Cipher.DECRYPT_MODE, _private);

            try {
                mensajePetición=((SealedObject)mensajePetición).getObject(cipher);
            } catch (IllegalBlockSizeException e) {
                e.printStackTrace();
            } catch (BadPaddingException e) {
                e.printStackTrace();
            }
            _key=new SecretKeySpec(((String)mensajePetición).getBytes(), "Blowfish");
        }
    }
}
```

2. We tell the client that we received the key using his key to encrypt the message and the server waits for an answer.

```
_conexión=true;
out.writeObject(blowfishCyp(new ClaveRecivida()));
mensajePetición = (SealedObject)in.readObject();
}
```

3. If we receive a cyphered text (SealedObject), the server decrypts it with the symmetric key.

```
if(( mensajePetición instanceof SealedObject )&&(!_conexión)){
    mensajePetición=blowfishDec((SealedObject)mensajePetición);
}
```

4. If it is a PeticiónConexión message (Connection Request) the server checks the data to be sure that is valid, then we send to that client the reply cyphered with the symmetric key.

```
if( mensajePetición instanceof PeticiónConexión ){
```

```

        if(_modelo.verificarUsuario(((PeticionConexion)mensajePeticion),this)){
            ((PeticionConexion)mensajePeticion).peticionAceptada(true);
        }
        else{
            ((PeticionConexion)mensajePeticion).peticionAceptada(false);
        }

        out.flush();
        out.writeObject(blowfishCyp(mensajePeticion));

```

5. If the connection does not succeed we we disconnect the client from the application.

```

        if(!((PeticionConexion)mensajePeticion).is_peticionAceptada()){
            _cliente.close();
        }

```

6. If the connection succeeded, the server adds the client to the connected client list and the server will process all the data sent by that client with his symmetric key.

```

        else{
            _nick=((PeticionConexion)mensajePeticion).getNick();
            _conectado=true;
            procesarMensajes();
        }
    }
}
catch (IOException e) {}
catch (NoSuchAlgorithmException e) {
    e.printStackTrace();
} catch (InvalidKeyException e) {
    e.printStackTrace();
} catch (NoSuchPaddingException e) {
    e.printStackTrace();
} catch (ClassNotFoundException e) {
    e.printStackTrace();
}
}
}

```

6.5.3 Common code between both applications :

1. SealedObject creation, in which we store the cyphered information from a Serializable object.

```

private SealedObject blowfishCyp(Serializable p) {
    try{

        Cipher cipher = Cipher.getInstance("Blowfish","BC");

        cipher.init(Cipher.ENCRYPT_MODE, _key);

        return new SealedObject(p,cipher);

    } catch (NoSuchAlgorithmException e) {
        e.printStackTrace();
    } catch (IllegalArgumentException e){
        e.printStackTrace();
    } catch (InvalidKeyException e) {
        e.printStackTrace();
    } catch (NoSuchPaddingException e) {
        e.printStackTrace();
    } catch (IllegalBlockSizeException e) {
        e.printStackTrace();
    } catch (NoSuchProviderException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
    return null;
}

```

2. This method replies with the decryption of the encrypted object in the SealedObject method.

```

private Object blowfishDec(SealedObject p) {
    try{

        Cipher cipher = Cipher.getInstance("Blowfish","BC");

        cipher.init(Cipher.DECRYPT_MODE, _key);

        return p.getObject(cipher);

    } catch (NoSuchAlgorithmException e) {
        e.printStackTrace();
    } catch (IllegalArgumentException e){
        e.printStackTrace();
    } catch (InvalidKeyException e) {
        e.printStackTrace();
    } catch (NoSuchPaddingException e) {

```

```

        e.printStackTrace();
    } catch (IllegalBlockSizeException e) {
        e.printStackTrace();
    } catch (NoSuchProviderException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    } catch (BadPaddingException e) {
        e.printStackTrace();
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
    return null;
}
}

```

6.5.4 Security problems found during the development

At the beginning we decided to use the RSA algorithm for the key exchange. The client was generating a public key and a private key. Then the client sends the public key to the server and the server encrypts a blowfish key. Then the server sends the encrypted Blowfish key to the client, the client decrypts the data and they establish the connection with the Blowfish key.

This situation can lead to a *man-in-the-middle attack*. As it was explained in Section 3.7.2, an attacker can use the client public key to cypher a blowfish key and send it to the client, therefore the attacker can have access to sensitive information. The client cannot detect the insecurity if the attacker acts as a server.

7 Conclusion

The symmetric algorithms encrypt and decrypt with the same key. The main advantages of the symmetric algorithms are their security and their encryption and decryption speed. The asymmetric algorithms encrypt and decrypt with different keys. The data is encrypted with a public key and they are decrypted with a private key, being this their main advantage. The asymmetric algorithms, also known as public key algorithms, they need bigger keys to reach a similar security level of a symmetric algorithm with a lower bit length key. They are much slower, therefore they cannot be used to encrypt huge amounts of data. As we can see in our study, the RSA algorithm, to encrypt a 512 bit message needs an average of 101288267 ns and the Blowfish needs only 2469,62 ns which is 41013 times faster.

On the other side, the public key algorithms allow us to create a secure channel in an unsecure medium, as we described in the discussion.

Therefore we have chosen to use a combination of an asymmetric algorithm and a symmetric algorithm. The symmetric algorithm will be used to send all the application data and the asymmetric algorithm is going to be used to set up the symmetric algorithm key at the beginning of the communication.

On the side of the symmetric algorithms, *Blowfish* appeared as the best candidate due to the performance and the big possibilities with the key length, up to 448 bits. *AES* is another good candidate because it is a standard and the security is well proved. Since we are going to set up a new key on each connection, *Blowfish* will be secure enough because each connection will not be long enough to break it if an attack appears in the following months. In this way we can use the advantage of *Blowfish* when we speak about performance. The big keys will be used to stay away from any brute force attack and this will be reinforced by the short-period connections. *Serpent* reaches our desired protection level but the lack of performance against *Blowfish* is a disadvantage. *DES* is a non-secure algorithm and it is slow, therefore it cannot be in our application. *Triple-DES* is secure but is less secure than *AES*, *Serpent* and *Blowfish* against brute force. It is also much slower.

For the asymmetric algorithm choice, *RSA* is much faster than *Diffie-Hellman* and since *Diffie-Hellman* can have problems with the *man-in-the-middle* attack, we have chosen *RSA*.

8 Future Work

This work can be used as a reference for people looking for the proper encryption algorithm concerning their needs.

For future work, more algorithms can be added to this study. Since some algorithms are faster in other programming languages, a good study can be a language-performance comparison of all the algorithms in many programming languages. With this information, not only the people programming in Java will take advantage of our work.

The testing application can be improved by adding only the CPU time consumption instead of the total amount of time spent. Memory usage could be also part of the performance/requirements to test.

Another interesting project could be the study of the hypothetical time in which the algorithms will be secure. The computer performance increase, possible theoretical attacks or newer improvements in computer science such as quantic computing. This project will be really interesting because in many occasions the data will be stored for long periods of time. This could be really sensitive information such as government secrets which they need to be secret, for example, 50 years. If they were encrypted with *DES*, at the beginning they were encrypted with a secure algorithm, but since nowadays that algorithm is no longer secure, people would be able to decrypt and access the original information.

References

- [1] Daniel Galin, *Software Quality Assurance : From theory to implementation*. pp. 39-41. Addison Wesley, 2004
- [2] Klaus Schmeh, *Cryptography and Public Key Infrastructure on the Internet* pp. 95-99. Wiley, 2001
- [3] Clifford Cocks. A Note on 'Non-Secret Encryption'. CESG Research Report, 20 November 1973
- [4] Klaus Schmeh, *Cryptography and Public Key Infrastructure on the Internet*. pp 59-64. Wiley, 2001
- [5] Antoine Joux, *Algorithmic Cryptanalysis*, pp. 157-163. CRC Press. 2009
- [6] Klaus Schmeh, *Cryptography and Public Key Infrastructure on the Internet* pp. 69-71. Wiley, 2001
- [7] John Talbot and Dominic Welsh, *Complexity and Cryptography. An introduction* pp. 117-118. Cambridge University Press
- [8] Klaus Schmeh, *Cryptography and Public Key Infrastructure on the Internet* pp. 75-82. Wiley, 2001
- [9] URL : <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>.
Last Access : 9/September/2010
- [10] Klaus Schmeh, *Cryptography and Public Key Infrastructure on the Internet* page 90. Wiley, 2001
- [11] B. Schneier *Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)*pp. 191-204. Fast Software Encryption, Cambridge Security Workshop Proceedings (December 1993), Springer- Verlag. 1994.
- [12] B. Schneier *Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)*pp. 191-204. Fast Software Encryption, Cambridge Security Workshop Proceedings (December 1994), Springer- Verlag. 1994.
- [13] Ross Anderson, Eli Biham, Lars Knudsen, *Serpent: A Proposal for the Advanced Encryption Standard*. 1998.
- [14] Klaus Schmeh, *Cryptography and Public Key Infrastructure on the Internet* pp. 93-95. Wiley, 2001
- [15] Serge Vaudenay *On the Weak Keys of Blowfish*. Ecole Normale Supérieure - DMI. 1995.
- [16] Vincent Rijmen, *Cryptanalysis and design of iterated block ciphers*, doctoral dissertation, October, 1997.
- [17] Eli Biham, Adi Shamir, *Differential Cryptanalysis of DES-like Cryptosystems*. The Weizmann Institute of Science. Department of Applied Mathematics. July, 1990.

- [18] Mitsuru Matsui, *Linear Cryptanalysis Method for DES Cipher*. Lecture Notes in Computer Science pp386-397. 1994.
- [19] Donald Davies, Sean Murphy, *Pairs and triplets of DES S-Boxes*. University of London. September, 1993.
- [20] Eli Biham, *How to Forge DES-Encrypted Messages in 2^{28} steps*. 1996.
- [21] Stefan Lucks, Bruce Schneier, Mike Stay, John Kelsey, David Wagner and Doug Whiting, *Improved Cryptanalysis of Rijndael* pp 213-230. 2000.
- [22] Daniel J. Bernstein, *Cache-timing attacks on AES*. 2005.
- [23] Bruce Schneier, *AES Timing Attack*. March, 2007.
- [24] Alex Biryukov, Nathan Keller, Orr Dunkelman, Dmitry Khovratovich and Adi Shamir, *Key Recovery Attacks of Practical Complexity on AES Variants With Up To 10 Rounds*. March, 2010.
- [25] Henri Gilbert, Thomas Peyrin, *Super-Sbox Cryptanalysis: Improved Attacks for AES-like permutations*. March, 2010.
- [26] Dhiman Saha, Dipanwita Roy Chowdhury, Debdeep Mukhopadhyay, *A Diagonal Fault Attack on the Advanced Encryption Standard*. December, 2009.
- [27] Tadayoshi Kohno, John Kelsey and Bruce Schneier, *Preliminary Cryptanalysis of Reduced-Round Serpent*. 2000
- [28] URL : <http://publib.boulder.ibm.com/infocenter/zos/v1r9/index.jsp?topic=/com.ibm.zos.r9.csfb>
Last access : September 8th, 2010
- [29] Stefan Lucks, *Attacking Triple Encryption*, pp 239–253. 1998
- [30] William C. Barker, *Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher*, NIST Special Publication 800-67. Revised 19 May, 2008
- [31] Ralph Merkle, Martin Hellman, *On the Security of Multiple Encryption*, Communications of the ACM, Vol 24, No 7, pp 465–467. July 1981.
- [32] Paul van Oorschot, Michael J. Wiener, *A known-plaintext attack on two-key triple encryption*, EUROCRYPT'90, LNCS 473, pp 318–325. 1990
- [33] U. Maurer, *Towards the equivalence of breaking the Diffie-Hellman protocol and computing discrete logarithms*, *Advances in Cryptology* pp 271-281. 1994
- [34] W. Diffie, P. C. van Oorschot, M.J. Wiener, *Authentication and Authenticated Key Exchanges* pp 107-125. 1992
- [35] URL : <http://en.wikipedia.org/wiki/S-box>
Date : 9/September/2010
- [36] URL : <http://upload.wikimedia.org/wikipedia/commons/thumb/6/6a/DES-main-network.png/222px-DES-main-network.png>
Last Access : 9/September/2010

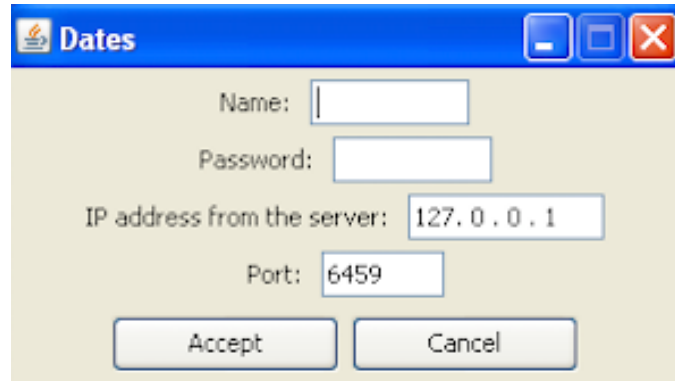
- [37] URL : <http://upload.wikimedia.org/wikipedia/commons/thumb/a/a3/DES-f-function.png/623px-DES-f-function.png>
Last Access : 9/September/2010
- [38] URL : <http://upload.wikimedia.org/wikipedia/commons/0/06/DES-key-schedule.png>
Last Access : 9/September/2010
- [39] URL : <http://upload.wikimedia.org/wikipedia/commons/thumb/d/de/Serpent-linearfunction.svg/301px-Serpent-linearfunction.svg.png>
Last Access : 9/September/2010
- [40] Based on the image from : URL : <http://upload.wikimedia.org/wikipedia/commons/thumb/1/13/Hellman-Schlüsselaustausch.svg/800px-Diffie-Hellman-Schlüsselaustausch.svg.png>
Date : 9/September/2010
- [41] URL : <http://upload.wikimedia.org/wikipedia/en/thumb/2/22/BlowfishFFunction.svg/250px-BlowfishFFunction.svg.png>
Last Access : 9/September/2010
- [42] URL : www.bouncycastle.org
Last Access : 9/September/2010
- [43] URL :

https://cds.sun.com/is-bin/INTERSHOP.enfinity/WFS/CDS-CDS_Developer-Site/en_US/-/USD/ViewProductDetail-Start?ProductRef=jce_policy-6-oth-JPR@CDS-CDS_Developer
Last Access : 9/September/2010

A User Manual

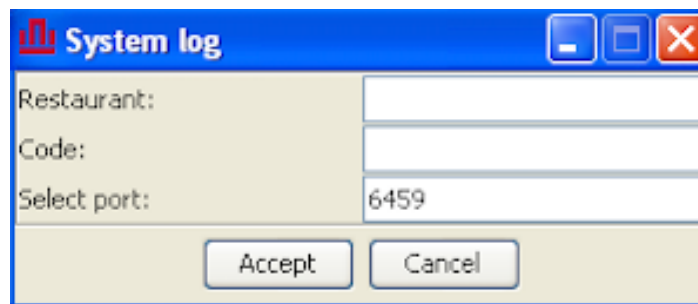
A.1 Login Windows

In Figure A.1.1 we can see the login screen for the clients and in Figure A.1.2 the one for the restaurants. The client is asked about an user name, password and the IP address and port for the server.

A screenshot of a Windows-style dialog box titled "Dates". It has a blue title bar with standard minimize, maximize, and close buttons. The dialog contains four text input fields: "Name:" (empty), "Password:" (empty), "IP address from the server:" (containing "127.0.0.1"), and "Port:" (containing "6459"). At the bottom are two buttons: "Accept" and "Cancel".

Name:	
Password:	
IP address from the server:	127.0.0.1
Port:	6459
<div>Accept Cancel</div>	

Figure A.1.1: Client Login Window

A screenshot of a Windows-style dialog box titled "System log". It has a blue title bar with standard minimize, maximize, and close buttons. The dialog contains three text input fields: "Restaurant:" (empty), "Code:" (empty), and "Select port:" (containing "6459"). At the bottom are two buttons: "Accept" and "Cancel".

Restaurant:	
Code:	
Select port:	6459
<div>Accept Cancel</div>	

Figure A.1.2: Restaurant Login Window

A.2 Main window : Restaurants

On the main window for the restaurants (Figure A.2.3) we can distinguish 3 sections :

1. Center part : All the requested information will appear in this section. There are 4 tab categories :
 - (a) Modification-Query tables : With 4 buttons to delete, modify or add discounts in the database. (Figure A.2.4)
 - (b) Client-Search Window : used to search clients with some specific attributes. (Figure A.2.5)
 - (c) Dish-search window : used to search dishes on the database with some specific attributes. (Figure A.2.6)
 - (d) Order-search window : used to search orders by month. (Figure A.2.7)
2. Bottom part : Here will appear the names of the tables of the database. If we select one, it will appear on the center part.
3. Right side : There are the buttons to request informs and listings (Figure A.2.8).

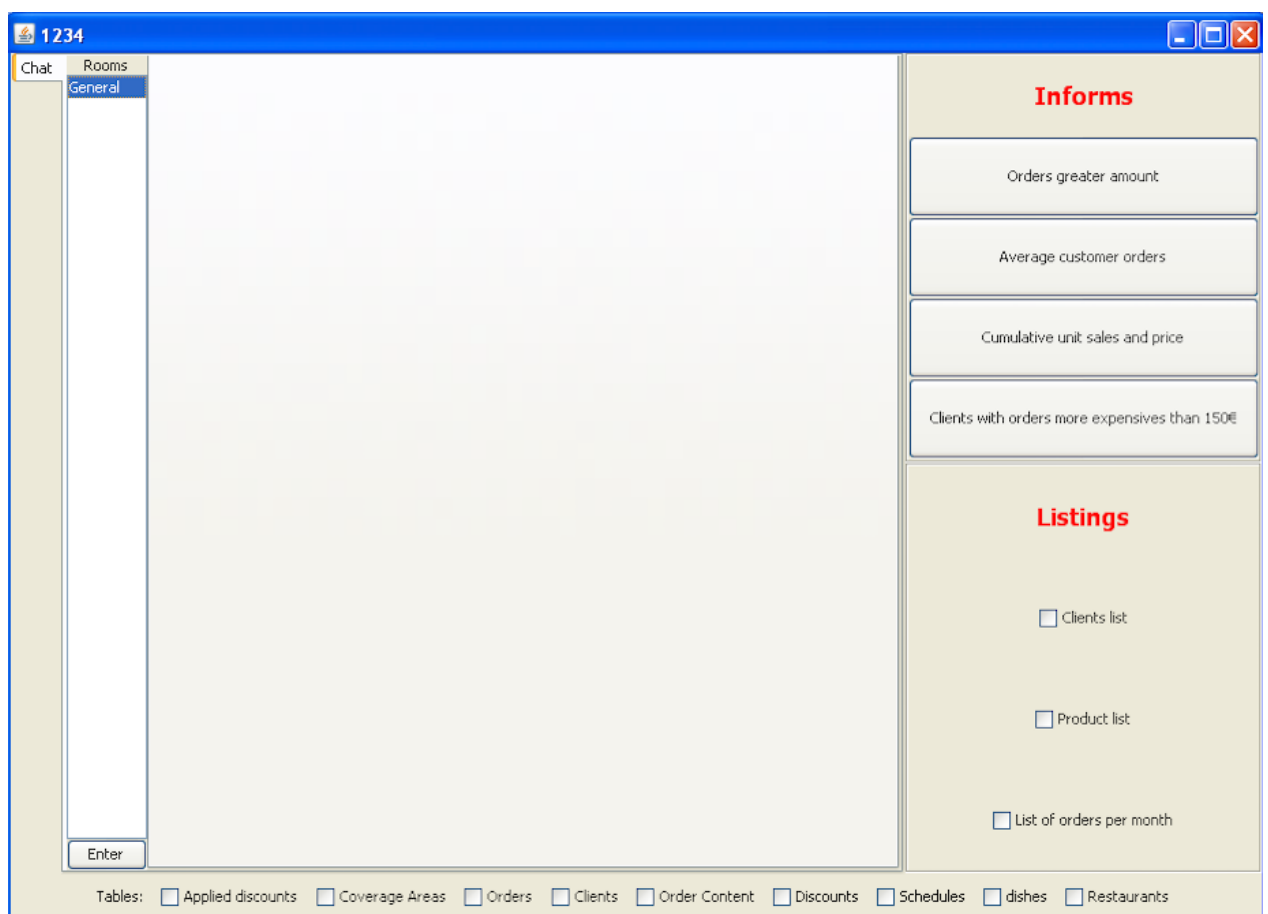


Figure A.2.3: Main Window : Restaurants

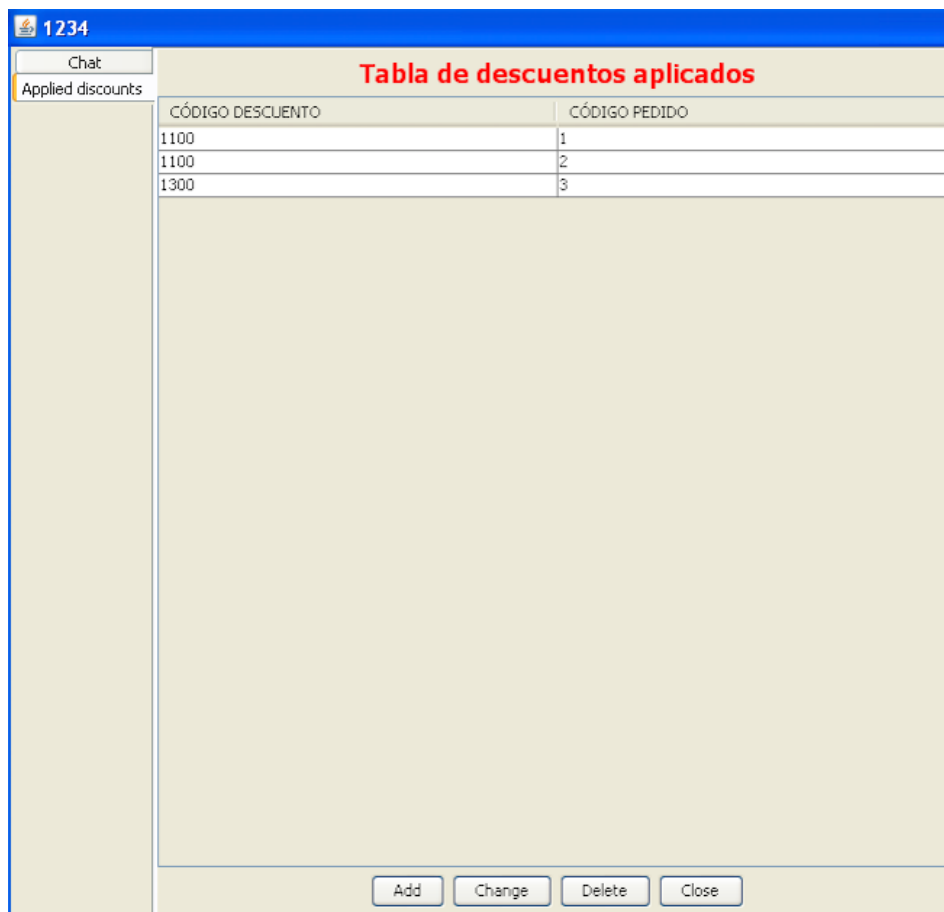


Figure A.2.4: Tab Window : Discount management

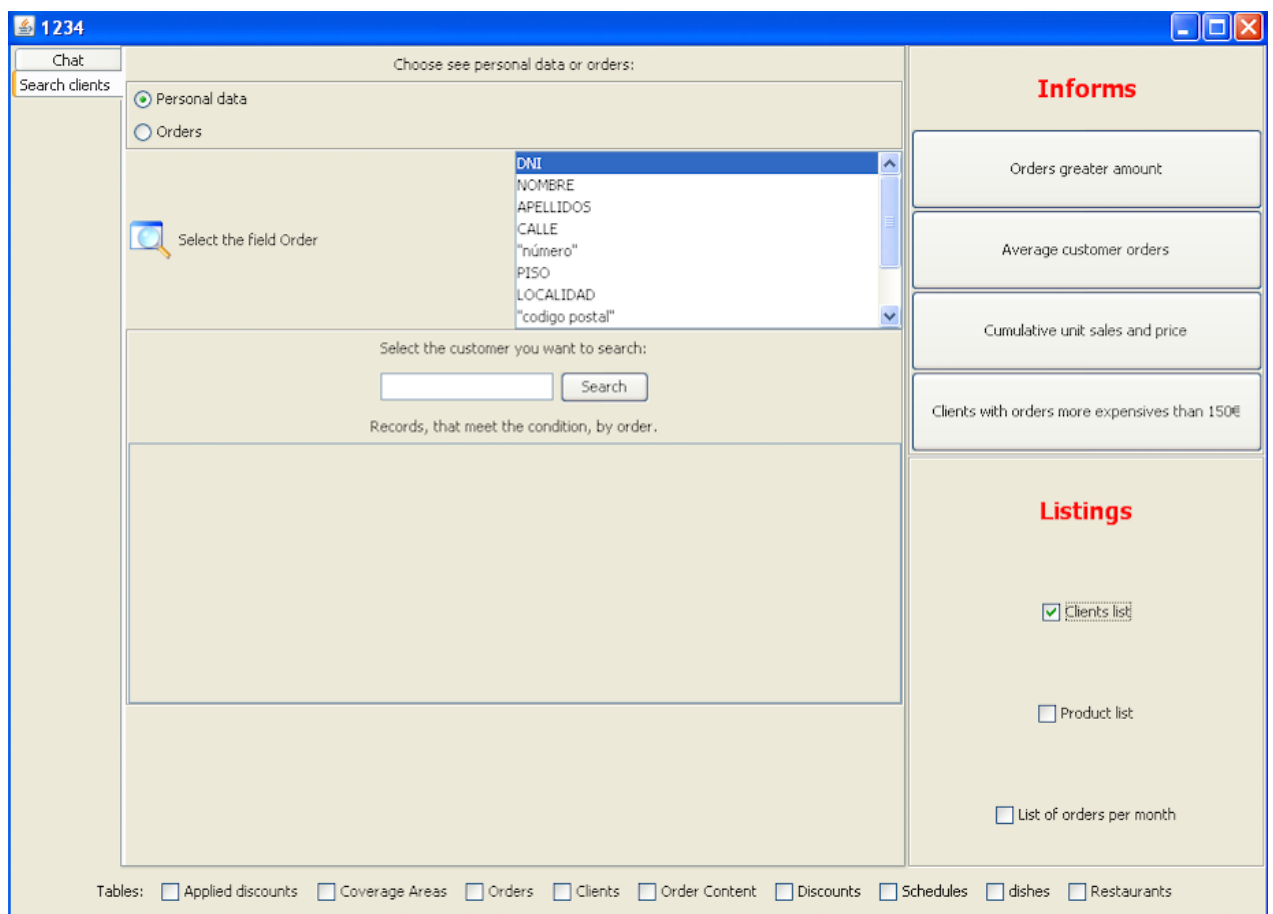


Figure A.2.5: Tab Window : Client management

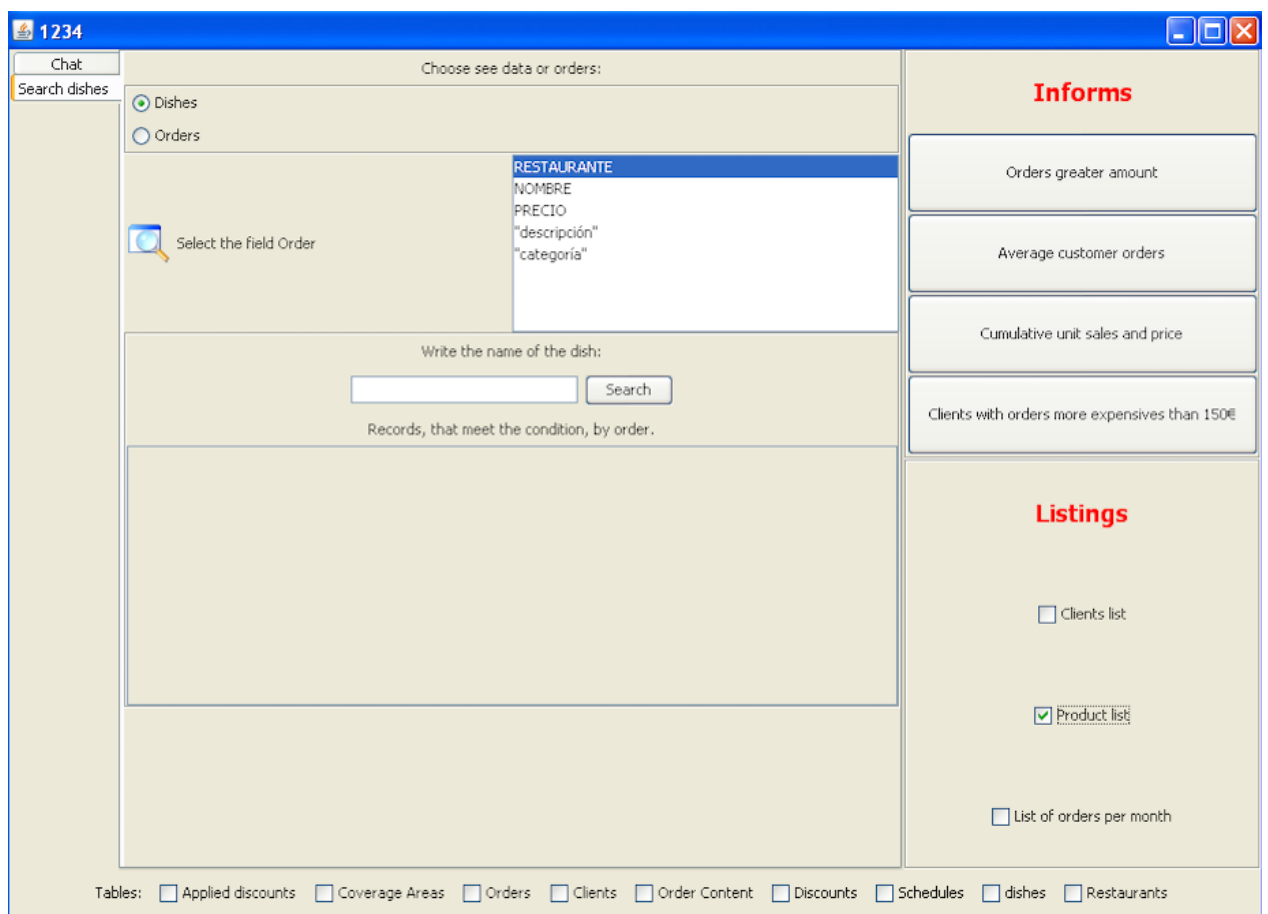


Figure A.2.6: Tab Window : Dish management

1234

Chat

Find orders

☒ January
 ☐ February
 ☐ March
 ☐ April
 ☐ May
 ☐ June
 ☐ July
 ☐ August
 ☐ September
 ☐ October
 ☐ November
 ☐ December

Search

Records, that meet the condition, by order.

Informs

Orders greater amount

Average customer orders

Cumulative unit sales and price

Clients with orders more expensive than 150€

Listings

☐ Clients list

☐ Product list

☒ List of orders per month

Tables:

☐ Applied discounts
 ☐ Coverage Areas
 ☐ Orders
 ☐ Clients
 ☐ Order Content
 ☐ Discounts
 ☐ Schedules
 ☐ dishes
 ☐ Restaurants

Figure A.2.7: Tab Window : Order search

Cumulative unit sales and price			
Inform Cumulative unit sales and price			
CÓDIGO RESTAU...	NOMBRE PLATO	UNIDADES	PRECIO ALCANZA...
1234	pizza arrabiata	3.0	17.85
1234	pizza margarita	2.0	
1234	pizza vegetal	2.0	0.0
2345	chana masala	4.0	37.8
2345	pollo tikka	3.0	0.0
3456	crunch-burger	1.0	0.0
3456	hot-burger	1.0	0.0
3456	vege-burger	5.0	0.0
5678	torta de carne espec	3.0	39.0225
5678	torta gallega	3.0	45.15

Figure A.2.8: List Window

A.3 Main window : Clients

On the main window for the clients (Figure A.3.9) we can distinguish 3 sections :

1. Center : All the requested information will be shown as we can see in Figure A.3.10.
2. Right : There is the shopping cart, where we can empty the cart, confirm the order or modify the order. If we use the right click button of our mouse, a menu will be shown to modify or delete any specific order. The watch with the actual time and the login time is also in that part of the screen.
3. Bottom : Here we can find the different lists we can request from our database.

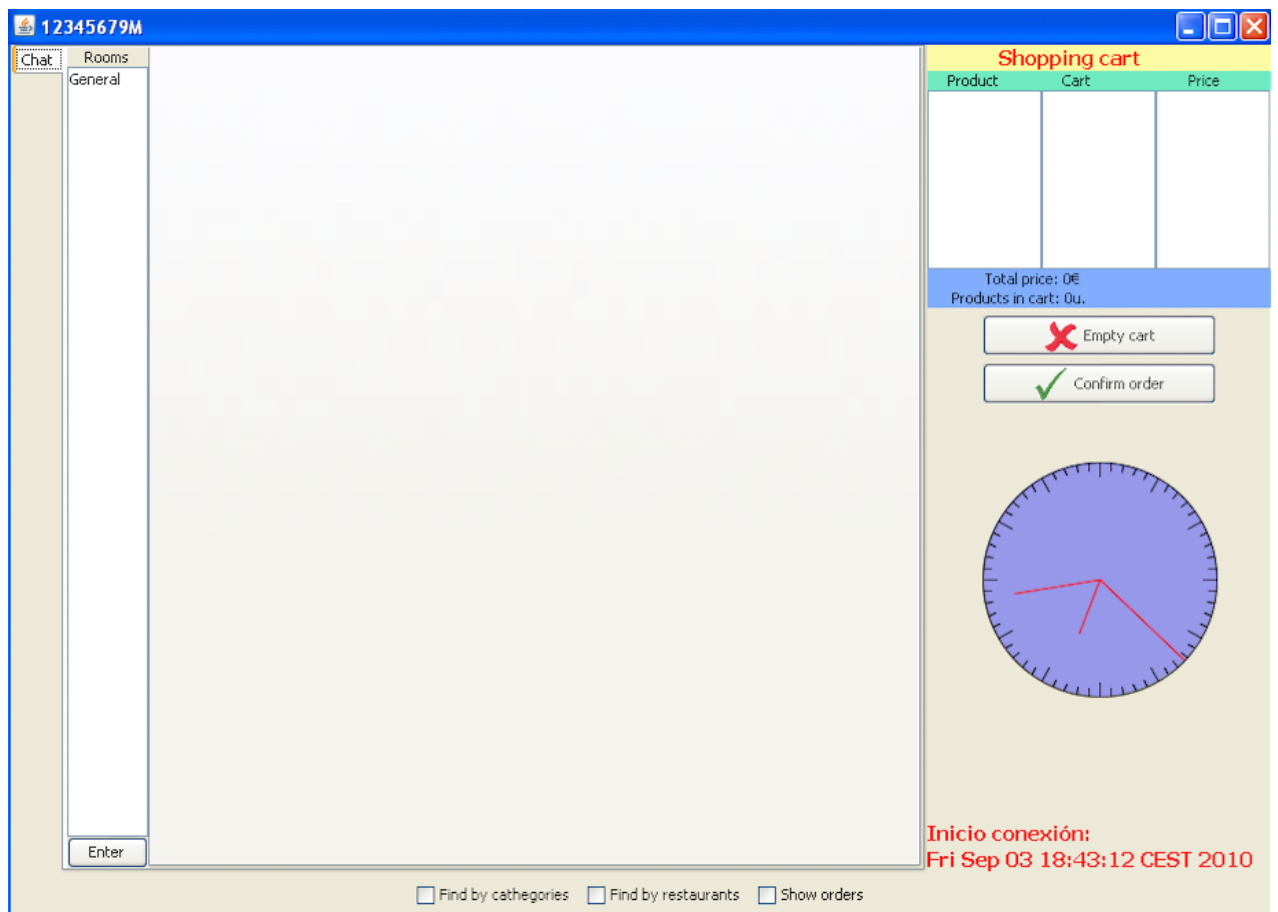


Figure A.3.9: Main Window : Clients

Chat

sin nombre

pizzahud

Templo hindú

burger-burger

telericatorta

Restaurante telericatorta

PLATO	CATEGORÍA	PRECIO	DESCRIPCION	CÓDIGO DEL RES...
torta de carne espec	carne	13.0075	torta de carne y qu...	5678
torta gallega	pescado	22.575	torta de pulpo	5678

Units: 1

Figure A.3.10: Main Window : Clients - Center

A.4 Main window : Restaurants and Clients

When the application starts, common on both applications, we can only see a chat button on the left and the chat room list (Figure A.4.11).

- Enter button : To get into the selected chat room
- Chat room list : If we double click in a selected chat room, we will enter the chat room. If we use the right click button, a meny will appear with several options depending if you are a client or a restaurant.(Figure A.4.12)
- Tab list : On the top of the window the different chat rooms where the user is logged in are represented each with a different tab. The cross on each tab allows the user to exit one specific chat room.

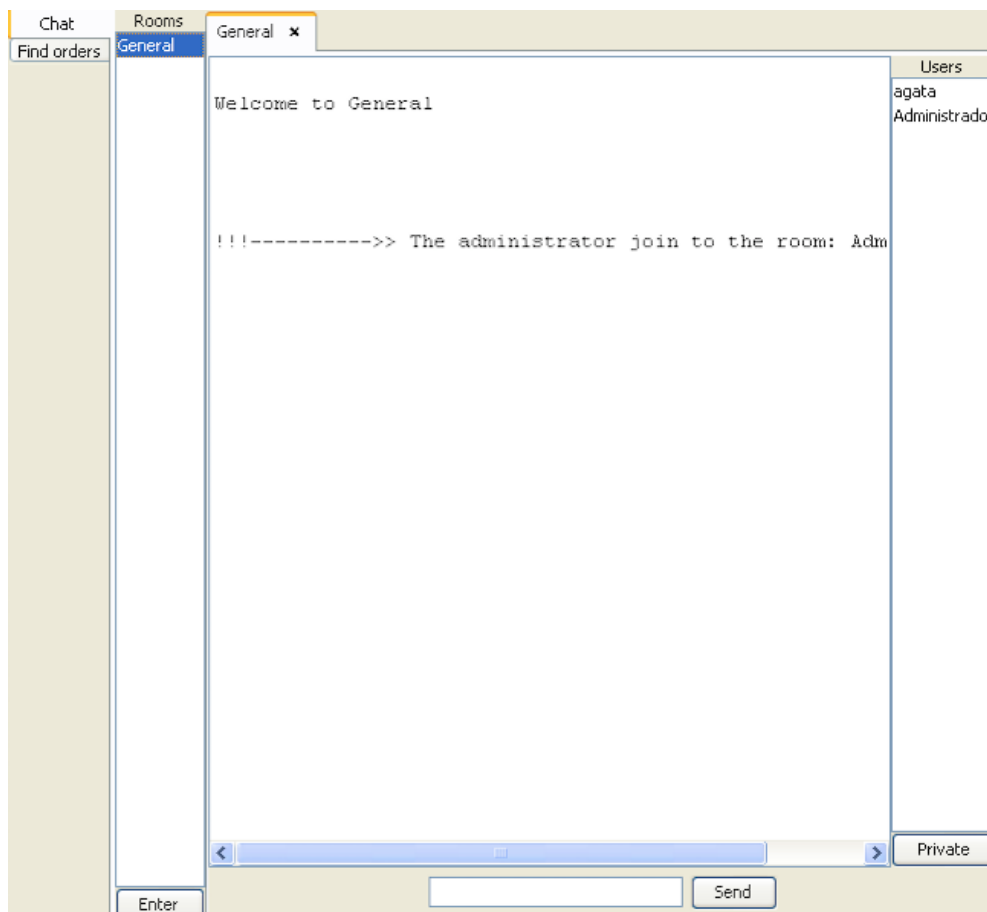


Figure A.4.11: Chat : Main Window



Figure A.4.12: Chat : Right click options

If you get into an specific chat room, (right and center part of Figure A.4.11), you can find the following buttons :

- Text field : to introduce the desired text to send
- Send button : to send the introduced text
- If we are in a non-private chat room :
 - User list : Here the user list on the same chat room is shown. If we double click on a nickname, a private chat room will start. If you use the right click button, a menu with different options will appear(Figure A.4.13)
 - Private button : It will create a private conversation with the selected chat user (equivalent to double click).

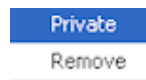


Figure A.4.13: Chat : Right click options II



Linnæus University

School of Computer Science, Physics and Mathematics

SE-351 95 Växjö / SE-391 82 Kalmar

Tel +46-772-28 80 00

dfm@lnu.se

[Lnu.se](http://lnu.se)